CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Spring 2008

Lecture 14— Makefiles & Compilation Management

Where are We

- Onto tools. . .
- Basics of make, particular the concepts
- Some fancier make features (revenge of funky characters)

Besides the slides and online Unix docs, the Stanford CSLib notes on Unix Programming Tools has a nice concise presentation of make and other tools:

- http://cslibrary.stanford.edu/107/UnixProgrammingTools.pdf

Onto tools

The language-implementation (preprocessor, compiler, linker, standard-library) is hardly the only useful thing for developing software.

The rest of the course:

- Tools (recompilation managers, version control, debuggers, profilers)
- Software-engineering issues
- A taste of C++
- Concurrency
- Societal implications

make

`make` is a classic program for controlling what gets (re)compiled and how. Many other such programs exist (e.g., `ant`, `maven`, "projects" in IDEs, ...)

`make` has tons of fancy features, but only two basic ideas:

1. Scripts for executing commands
2. *Dependencies* for avoiding unnecessary work

To avoid "just teaching make features" (boring and narrow), let's focus more on the concepts...

Build scripting

Programmers spend a lot of time "building" (creating programs from source code)

- Programs they write
- Programs other people write

Programmers automate repetitive tasks. Trivial example:

```
gcc -Wall -g -o myprog foo.c bar.c baz.c
```

If you:

- Retype this every time: "shame, shame"
- Use up-arrow or history: "shame" (retype after logout)
- Have an alias or bash script: "good-thinkin"
- Have a Makefile: you're ahead of us

"Real" build processes

On larger projects, you can't or don't want to have one big (set of) command(s) that redoes everything every time you change anything.

1. If `gcc` didn't combine steps behind your back, you could need to preprocess and compile each file, then call the linker.
2. If another program (e.g., `sed`) created some C files, you would need an "earlier" step.
3. If you have other outputs for the same source files (e.g., `javadoc`), it's unpleasant to type the source files multiple times.
4. If you want to distribute source code to be built by other users.
5. If you have $\mathbf{10^5}$ to $\mathbf{10^7}$ lines of source code, you don't want to recompile them all every time you change something.

A simple script handles 1–4 (use a variable for the filenames for 3), but 5 is trickier.

Recompilation management

The "theory" behind avoiding unnecessary compilation is a "dependency dag":

- To create a target t, you need sources s_1,s_2,...,s_n and a command c (that directly or indirectly uses the sources)
- If t is *newer* than every source (file-modification times), assume there is no reason to rebuild it.
- Recursive building: If some source s_i is itself a target for some other sources, see if it needs to be rebuilt. Etc.
- Cycles "make no sense"

Theory applied to C

Another whole lecture on *linking* is in our future, but here is what you need to know today for C:

- Compiling a `.c` creates a `.o` and depends on all included files (recursively/transitively).
- Creating an executable ("linking") depends on `.o` files.
- So if one `.c` file changes, just need to recreate one `.o` file and relink.
- If a header-file changes, may need to rebuild more.
- Of course, this is only the simplest situation.

An algorithm

What would a program (e.g., a shell script) that did this for you look like? It would take:

- a bunch of triples: target, sources, command(s)
- a "current target to build"

It would compute what commands needed to be executed, in what order, and do it. (It would detect cycles and give an error.)

This is exactly what programs like `make`, `ant`, and things integrated into IDEs do!

make basics

The "triples" are typed into a "makefile" like this:

```
target: sources
        command
```

Example:

```
foo.o: foo.c foo.h bar.h
       gcc -Wall -o foo.o -c foo.c
```

Syntax gotchas:

- The colon after the target is required.
- Command lines must start with a **TAB NOT SPACES**
- You can actually have multiple commands (executed in order); if one command spans lines you must end the previous line with \.
- Which shell-language interprets the commands? (Typically bash, to be sure set the SHELL variable in your makefile.)

Using `make`

At the prompt:

```
prompt% make -f nameOfMakefile aTarget
```

Defaults:

- If no `-f` specified, use a file named `Makefile`.
- If not target specified, use the first one in the file.

Together: I can download a tarball, extract it, type `make` (four characters) and everything should work.

Actually, there's typically a "configure" step too, for finding things like "where is the compiler" that *generates* the `Makefile` (but we won't get into that).

Basics Summary

So far, enough for homework 4 and basic use.

- A tool that combines scripting with dependency analysis to avoid unnecessary recompilation.
- Not language or tool-specific: just based on file-modification times and shell-commands.

But there's so much more you want to do so that your Makefiles are:

- Short and modular
- Easy to reuse (with different flags, platforms, etc.)
- Useful for many tasks
- Automatically maintained with respect to dependencies.

Also, reading others' makefiles can be tough because of all the features: see `info make` or entire books.

Precise review

A Makefile has a bunch of these:

```
target: source1 ... sourcen
        shell_command
```

Running `make target` does this:

- For each source, if it is a target in the Makefile, process it recursively
- *Then:*
  - If some source does not exist, error.
  - If some source is newer than the target (or target does not exist), run `shell_command` (presumably updates `target`, but that is up to you).

make variables

You can define variables in a Makefile. Example:

```
CC = gcc
CFLAGS = -Wall
foo.o: foo.c foo.h bar.h
	$(CC) $(CFLAGS) -c foo.c -o foo.o
```

Why do this?

- Easy to change things once and affect many commands.
- Can change variables on the command-line (overrides definitions in file). (For example `make CFLAGS=-g`.)
- Easy to reuse most of a Makefile from one "homework" to the next.
- Can use conditionals to set variables (using inherited environment variables):

make conditionals

```
EXE=
ifdef WINDIR # assume we are on a Windows machine
  EXE=.exe
endif
myprog$(EXE): foo.o bar.o
      $(CC) $(CFLAGS) -o myprog$(EXE) foo.o bar.o
```

Other forms of conditionals exist (e.g., are two strings equal)

more variables

It's also common to use variables to hold list of filenames:

```
OBJFILES = foo.o bar.o baz.o
myprog: $(OBJFILES)
	gcc -o myprog $(MYOBJFILES)
clean:
	rm $(OBJFILES) myprog
```

`clean` is a convention: remove any generated files, to "start over" and have just the source.

It's "funny" because the target doesn't exist and there are no sources, but that's okay:

- If target doesn't exist, it must be "remade" so run the commands
- These "phony" targets have several uses, another is an "all" target:

"all" example

```
all: prog B.class someLib.a # notice no commands this time

prog: foo.o bar.o main.o
	gcc -o prog foo.o bar.o main.o

B.class: B.java
	javac B.java

someLib.a: foo.o baz.o
	ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
	gcc -c -Wall foo.c

... (similar targets for bar.o, main.o, baz.o) ...
```

Revenge of funny characters

UNIX hackers just can't get enough of funny metacharacters can they?

In commands:

- `$@` for target
- `$^` for all sources
- `$<` for left-most source
- ...

Examples:

```
myprog$(EXE): foo.o bar.o
	$(CC) $(CFLAGS) -o $@ $^

foo.o: foo.c foo.h bar.h
	$(CC) $(CFLAGS) -c $<
```

More fancy stuff

- There are a lot of "built-in" rules. E.g., `make` just "knows" to create `foo.o` by calling `$(CC) $(CFLAGS)` on `foo.c`. (Opinion: more confusing than helpful. YMMV)
- There are "suffix" rules and "pattern" rules. Example:

```
%.class: %.java
        javac $<    # Note we need $< here
```

- Remember you can put any shell command on the command-line, even whole scripts
- You can repeat target names to add more dependencies (useful with automatic dependency generation).

Often this stuff is more useful for reading makefiles than writing your own (until some day...)

Dependency generation

So far, we are still listing dependencies manually, e.g.:

```
foo.o: foo.c foo.h bar.h
```

If you forget, say, `bar.h`, you can introduce subtle bugs in your program (or if you're *lucky*, get confusing errors).

This is not make's problem: It has no understanding of different programming languages, commands, etc., just file-mod times.

But it does seem too error-prone and busy-work to have to remember to update dependencies, so there are often language-specific tools that do it for you...

Dependency-generator example

`gcc -M`

- Actually lots of useful variants, including `-MM` and `-MG`. See `man gcc`
- Automatically creates a rule for you.
- Then `include` the resulting file in your Makefile.
- Typically run via a phony `depend` target, e.g.:

```
depend: $(PROGRAM_C_FILES)
        gcc -M $^
```

- The program `makedepend` combines many of these steps; again it is C-specific but some other languages have their own.

Build-script summary

Always script complicated tasks.

Always automate "what needs rebuilding" via dependency analysis.

`make` is a text-based program with lots of bells and whistles for doing this. It is not language-specific. Use it.

With language-specific tools, you can automate dependency generation.

`make` files have this way of starting simple and ending up unreadable. It is worth keeping them clean.

There are conventions like `make all` and `make clean` common when distributing source code.