CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Autumn 2007

Lecture 14— Makefiles continued; Breakpoint debugging & gdb

Where are We

- Basics of make, particular the concepts (last lecture)
- Some fancier make features (revenge of funky characters)
- Start debuggers, particular `gdb`

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

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

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.

An execution monitor?

What would you like to “see from” and “do to” a running program?

Why might all that be helpful?

What are reasonable ways to debug a program?

A “debugger” is a tool that lets you stop running programs, inspect (sometimes set) values, etc.

- A “CAT scan” for observing executing code

Issues

- Source information for compiled code. (Get compiler help.)
- Stopping your program too late to find the problem. (Art.)
- Trying to “debug” the wrong algorithm.
- Trying to “run the debugger” instead of understanding the program.

It’s an important tool.

Debugging C vs. Java

- Eliminating crashes does not make your C program correct.
- Debugging Java is “easier” because crashes and memory errors do not exist.
- But programming Java is “easier” for the same reason!

gdb

`gdb` (Gnu debugger) is on `attu` and supports several languages, including C compiled by `gcc`.

Modern IDEs have fancy GUI interfaces, which help, but concepts are the same.

Compiling with debugging information: `gcc -g`

- Otherwise, gdb can tell you little more than the stack of function calls.

Running `gdb`: `gdb` *executable*

- Source files should be in same directory (or use the `-d` flag).

At prompt: `run` *args*

Note: You can also inspect core files, which is why they get saved. (Mostly useful for analyzing crashed programs after-the-fact, not for systematic debugging.)

Basic functionality

- backtrace
- frame, up, down
- print *expression*, info args, info locals

Often enough for "crash debugging"

Also often enough for learning how "the compiler does things" (e.g., stack direction, malloc policy, ...)

Breakpoints

- break *function* (or line-number or ...)
- conditional breakpoints (break XXX if expr)
  1. to skip a bunch of iterations
  2. to do assertion checking
- going forward: `continue`, `next`, `step`, `finish`
  - Some debuggers let you "go backwards" (typically an illusion)

Often enough for "binary search debugging"

Also useful for learning program structure (e.g., when is some function called)

Why not skim the manual for other features.

Advice

Understand what the tool provides you.

Use it to accomplish a task, for example “I want to know the call-stack when I get the NULL-pointer dereference”

Optimize your time developing software.

- Think of debugging as a systematic experiment to discover what’s wrong — not a way to randomly poke around.

Use development environments that have debuggers?

See also: `jdb` for Java (on `attu`)