Lab 8

Out: Thursday, March 24, 2005

What you'll learn.

In this lab, you'll write your own Makefile, and hopefully see why they can be a little tricky to fix. You'll also learn how to debug your programs more effectively and write more efficient code.

How you'll do it.

So far, you've used make a lot, but how it works might still be a little mysterious. So you'll build up a Makefile, step by step, to see how it's working. You'll also be taking a TA-provided source and Makefile (`/course/cs034/pub/lab8/rot13/*`) and using `gdb`, `valgrind`, and `gcov` to debug it. Additionally, you'll learn how to better use your editor (either `xemacs` or `vim`) to quickly navigate, produce, and format code.

1 Task 1: Writing your own Makefile

Makefiles work off of dependencies. They're composed of a series of rules of the form

```
<target> : <list of dependencies>
<tab character> <line to execute>
```

When `make` runs on a target, it compares the timestamp of the target to the timestamp of each item in its list of dependencies. If any of the dependencies have been modified more recently than the target, it will look for another rule for that out-of-date dependency, then execute the commands on the lines below (these have to be indented by a tab character, or `make` will complain). You can specify which target to evaluate on the command line; if you don't, `make` will run the target "all", or if that doesn't exist, then the first target in the file.

Task:

- Copy over the files from /course/cs034/pub/lab8 into your directory. Both first.c and second.c depend on library.h, so you'll need to compile them together.
- Write a simple Makefile to make both programs. Each executable will depend on two .o files, and each .o file depends on a .c file. (Yes, there is an easier way to do this, but do it the explicit way first).
- You can add a "@echo" line in each rule, also indented by a tab, that will print out a string when that rule executes. Using these print lines, see what happens when you update the source files and remake.

2 Task 2: Implicit Rules

It's not fun to write out a rule for each .c file, especially when they're so repetitive. Fortunately, `make` does it for you! You can pattern-match the extensions of files using the % sign. So, to compile .o files from .c files, you could write a rule like this:

```
%.o : %.c
    // compile the files.
```

You can access the name of the source file with the macro "$<", and the name of the target file with "$@".

Task: Create an implicit rule in your Makefile to compile the .c files to .o files automatically. Again, add a print line with @echo to see when this rule executes.

Actually, `make` does this particular rule (and other common ones) automatically. To find out which, you can `man make` or google Makefile. So even without this rule, the .c files will be compiled, but how do you specify flags to the compiler? You can do this with macros, of the form

```
NAME = value
```

The implicit rule that make uses for making .o's from .c's uses the compiler defined as CC, and the flags defined as CFLAGS. These can be specified at the top of the file, on the command line, or in the environment. You can also access these values by using $(NAME) at any point in the Makefile.

3 Task 3: Fun with Debugging

`make` can be used for more than just compiling code, of course. You can put arbitrary commands in the lines to execute for each target. For example, if you wanted to view an image you'd just generated, whose path was a line in the file "image", you could include a line like this in your Makefile:

```
image:
    xv `cat image`
```

But sometimes `make`'s rules act in unexpected ways.

> **Task:** Try running `make` with the Makefile fragment given above, and fix the problem.

Task 4: Using your editor

Being a good programmer isn't all about writing quality code – sometimes, it can be about writing quality code *fast*. One easy way to become a faster coder is to learn an editor and learn it well. Knowing your way around your editor allows you to quickly navigate where you need to be, move around blocks of code, and let it help you notice typos or other syntax errors before even compiling. Editors like `vim` and `emacs` allow you to do all this without moving your hands from the keyboard, saving you large amounts of time over the course of a big project.

To get started, copy the contents of /course/cs034/pub/lab8/rot13/ into your lab directory. Open `rot13.c` in your preferred editor (preferably `vim` or `xemacs` for the purpose of this lab).

First, you'll notice that the file hasn't been formatted nicely at all. Follow the instructions below to fix this:

vim

`Esc` (command-mode)
`gg` (go to beginning of file)
`v` (visual selection mode)
`shift-g` (go to end of file)
`=` (indent selected region)

xemacs

`C-x h` (mark-whole-buffer, selects all)
`M-C \` (indent-region)
`C-x C-s` (save-buffer)

(where C is control, M is Alt)

Now, try compiling by typing `make`. You'll see an error because the `rot13` function is defined below `main` and has no prototype. Rather than implementing a prototype, take this opportunity to practice cutting and pasting without using the mouse:

vim	xemacs
`Esc` (command-mode)	navigate to above `rot13` function
navigate to above `rot13` function (line 37)	`C-space` (set selection start)
`v` (enter visual selection mode)	navigate to below `rot13` function
`shift-g` (go to end of file)	`C-w` (kill-region)
`x` (cut selected region)	navigate to above `main` function
navigate to above `main` function	`C-y` ("yank" (paste))
`p` (paste)	
`:w` (save)	

Save `rot13.c` without using your mouse, and check that `make` runs without any warnings. If your window manager is set up for "focus follows mouse", try using `shift-alt-arrowkeys` to switch focus between your editor and your terminal without having to use your mouse.

Task 5: Debug ROT13

After compiling `rot13`, give it a try, and notice that it has a segmentation fault upon trying to encode any text. In order to track it down, run `gdb rot13`. At the `(gdb)` prompt, type `run` to start your program.

You'll notice that your program still has a segmentation fault, but this time `gdb` breaks as the error occurs and prints the line that triggered the invalid memory access. Use gdb's `print` command to inspect the value of the variables involved. In more complicated programs, the `bt` (backtrace) command may be useful to see the way in which your program got to the crash point.

> **Task:** Use `gdb` to figure out why `rot13` is causing a segmentation fault. Fix the bug. For now, don't worry about memory leaks.

After fixing the above bug, you should be able to run `rot13` without a segfault; however, this does not mean that your program is bugfree. In order to check `rot13` for other types of bugs that may not always be fatal, we'll use `valgrind`, a memory debugging tool. Valgrind is essentially an x86 emulator that runs your program in a safe "sandbox" and logs memory problems. In order to run your program inside `valgrind`, simply type `valgrind rot13`.

> **Task:** Use the output of `valgrind` to find and fix another bug in the `rot13` function. `valgrind` should report a `Conditional jump or move depends on uninitialized values` warning.

The fact that the bug above probably didn't cause a crash is not unusual. Often you will write software that will work "by coincidence" for quite some time until you happen to feed it some data that exposes the lurking memory bug.

In addition to memory bugs that can cause crashes, another type of bug that `valgrind` can help detect is the "memory leak." Memory leaks occur when you `malloc` a block of memory but never remember to `free` it. In a short program like `rot13` this isn't a big deal, but in a server that should run for months without restarting, memory leaks are very serious.

Task: Run `valgrind --leak-check=yes rot13` in order to see a summary of the memory leaks in your program. `valgrind` helps you solve them by letting you know where the offending memory blocks were allocated. Use the output of `valgrind` with leak-checking enabled to fix the memory leak in `rot13`

Coverage Testing

Now that `rot13` isn't crashing, how can you be reasonably sure that you've actually tested all of its capabilities? The answer can be found in `gcov`, a "coverage testing" tool. `gcov` will tell you how many times each line of your program executed, and more importantly, which lines of your program have not been run at all.

In order to run your program under `gcov`, you'll need to pass some additional flags to `gcc`. In order to do this, open up your `Makefile` in either `vim` or `xemacs` and modify the `CFLAGS` variable to include `-fprofile-arcs -ftest-coverage`. Save your Makefile, `make clean`, and then `make`. Run `rot13` as usual and type in a test word in all lower case. Exit the program by entering a "." on a blank line. Now run `gcov rot13.c`.

`gcov` will analyze and tally the way `rot13` ran and create `rot13.c.gcov`. Open this file up in an editor and look for lines marked `#####`. These are lines of your program which did not run. Since you typed in a word without any lower case letters, you should notice this marking next to `out[i] = toupper(out[i]);`.

Task: Run `rot13` several times with different types of input, re-run `gcov`, and examine `rot13.c.gcov` until you can verify that every line of your program has been tested.