Problem Set 1: C

out of 55 points

due by 7:00 P.M. on Friday, 5 October 2007

Be sure that your code is thoroughly commented
to such an extent that lines' functionality is apparent from comments alone.

Goals.

The goals of this problem set are to:

- Start turning you into a Linux wonk.
- Have you solve some problems in C.
- Introduce you a bit early to bitwise operations.

Recommended Reading.

Per the syllabus, no books are required for this course. If you feel that you would benefit from some supplementary reading, though, below are some recommendations.

- Sections 1 – 7, 9, and 10 of `http://www.howstuffworks.com/c.htm`.
- Chapters 1 – 6 and 12 of *Programming in C*.

Academic Honesty.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed (*e.g.*, by some problem set or the final project). Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, magazine, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student.

Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this class that you have submitted or will submit to another. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

All forms of cheating will be dealt with harshly.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the staff.

Getting Started.

0. Visit the forum of **Final Project Ideas** on the course website's **Bulletin Board** and read over the ideas that have already been posted. Feel free to post zero or more of your own!

1. Assuming you already configured your FAS account for use in this course for Problem Set 0, SSH to `nice.fas.harvard.edu` and execute the following sequence of commands.[1]

```
cd ~/cs50
mkdir ps1
cd ps1
```

Your prompt should now resemble the below.

```
username@nice (~/cs50/ps1):
```

Within your prompt's parentheses, recall, is a reminder of your current working directory. Know that `..` represents your current working directory's "parent directory." Go ahead and type the command below to go "up one directory" (*i.e.*, backwards).

```
cd ..
```

Your prompt should now resemble the below.

```
username@nice (~/cs50):
```

It's also worth knowing that `.` represents your current working directory. Go ahead and type the command below.

```
cd .
```

Your prompt should still look the same. Pointless, eh? Trust us, though, `.` does have its uses. In fact, you may have seen me start typing commands like the below in lecture only to stop myself and delete the leading dot and slash.

```
./a.out
```


[1] Recall that you created your `ps0/` directory with a somewhat different command for Problem Set 0. Needless to say, there are multiple ways to get the job done!

The above command means that I want to execute a program called `a.out` that's in my current working directory. On most Linux systems, . is not in your `$PATH`, an "environment variable" whose value is a colon-separated list of directories that your shell should check when you execute commands. To execute a program in your current working directory on such systems, you thus need to specify the program's location. Hence the command above. For simplicity, we've added . to your `$PATH` for CS 50. Execute the command below to see what else is in your `$PATH`.

```
echo $PATH
```

Better yet, execute the command below to see all your environment variables.

```
setenv
```

Kind of messy, yes? Try

```
setenv | sort
```

instead, which "pipes" the output of `setenv` into `sort` as input. Want to see where most of those environment variables came from? Take a look at the contents of `~/.cshrc`, a file that's loaded by your shell upon login. (Another such file is `~/.login`.) Rather than use Nano (or even Vim or Emacs), try the command below instead.

```
less ~/.cshrc
```

Or, instead of `less`, try `cat` or `more`. To learn more about `less` (ha), execute the command below to pull up the command's "man page."

```
man less
```

Alternatively, execute the command below to see instructions built into the command itself.

```
less --help
```

You can almost always learn more about commands using `man` or `--help` in this manner. Worst case, Google and the staff are your friends. Notice, now, that `~/.cshrc` is "sourcing" (*i.e.*, loading) `~cs50/pub/etc/cs50.cshrc`. It's by way of that file that we have configured your account for CS 50. Take a look at it and see how much of its functionality you can infer. Feel free to ask questions about it via the Bulletin Board. If you want to learn more about the "language" in which it's written, Google "shell scripts" or the like. Know that the shell you're using on `nice.fas.harvard.edu` is called `tcsh`.

Anyhow, navigate your way back to `~/cs50/ps1/`. (Remember how?) All of the work that you do for this problem set must ultimately reside in that directory for submission.

Incidentally, know that if you ever get "lost" within your account, simply type the below to be whisked back to the "root" of your home directory.

```
cd
```

Alternatively, you could type the below.

```
cd ~
```

Or the below.

```
cd $HOME
```

There's one of those environment variables again. In fact, let's conclude with a question about the first environment variable you met. In turns out that adding `.` to one's `$PATH`, however convenient, does poses a security risk. In an ASCII file called `hypothesis.txt`, hypothesize in one or more sentences why that might be.

2. (5 points.) We've all printed things out before (including, perhaps, this problem set), but have you ever thought about how the process works? In a paragraph of your own words in a file called `printers.txt`, how do laser printers and, by contrast, inkjet printers work?[2]

Let's C What You Can Do.

3. (5 points.) Let's get you warmed up. Rather than use Nano, you might want to teach yourself how to use Emacs or Vim, both of which are more powerful text editors than Nano. Once you've decided on an editor, create within your `~/cs50/ps1/` directory a file called `hello.c`. Proceed to write your own version of "hello, world." It suffices to re-type, character for character, Week 1's `hello1.c`, but do at least replace "hello, world" with your own argument to `printf`.

   Once done with your recreation, hit ctrl-x to save, followed by Enter, and you should be returned to your prompt. Proceed to execute the command below.

```
gcc hello.c
```

   If you've made no mistakes, you should see but another prompt. If you've made some mistake, you'll instead see some error message. Even if cryptic, think about what it might mean, then go fix your mistake.

[2] For this question, you're welcome to consult *How Computers Work*, Google, Wikipedia, a friend, or anyone else, so long as your words are ultimately your own!

Once your code is correct and compiles successfully, look for your program in your current working directory by typing the following command.

```
ls -l
```

More than just list the contents of your current working directory, this command lists their sizes, dates and times of creation, and more. The output you see should resemble the below.

```
-rwx------ 1 username student 6886 2007-09-27 18:04 a.out*
-rw------- 1 username student  373 2007-09-27 18:03 hello.c
```

The `-l` is a "switch," which controls the behavior of `ls`. To look up more switches for `ls` (and its documentation in general), pull it up its man page.

Let's now confirm that your program does work. Execute the command below.

```
a.out
```

You should see your greeting. Before moving on, let's give your program a more interesting name than `a.out`. Go ahead and execute the following command.

```
gcc -o hello hello.c
```

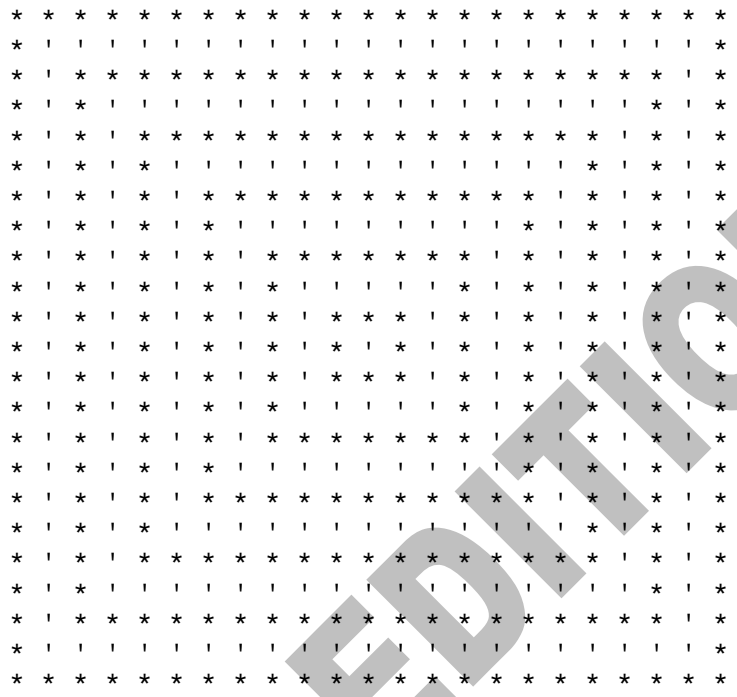
Needless to say, `-o` is but a switch for `gcc`. The effect of this switch is to name `gcc`'s output `hello` instead of `a.out`. Let's now get rid of your first compilation. To delete `a.out`, execute the following command.

```
rm a.out
```

If prompted to confirm, hit y followed by Enter.

Welcome to Linux and C.

4. (15 points.) Let us refer to the below ASCII art as "nested squares" of apostrophes and asterisks.



Write, in `squares.c`, a program that produces nested squares. This program must first prompt the user for the length (in characters) of the outermost square's edge. (The above example happens to use 23.) It must then ask the user for the character to use for the outermost (and every other) square. Assuming the user provided a length of 3 or greater, it must also then prompt the user for the other character to use. It should then produce the nested squares. Assume that the user's terminal window is exactly 80 characters wide by 24 characters tall. (Best to ensure that your own window boasts at least those dimensions.) So that a blinking prompt still fits on the screen after the nested squares' generation, demand that the user provide a positive, integral length no greater than 23. We leave it to you, of course, to figure out how to produce these squares dynamically!

Speaking of ASCII art, execute the command below when you get a chance. (Telnet is similar in spirit to SSH, but it's not encrypted.) Wow, right?

```
telnet towel.blinkenlights.nl
```

5. (15 points.) Toward the end of World 1-1 in Nintendo's Super Mario Brothers, Mario must ascend a "half-pyramid" of blocks before leaping (if he wants to maximize his score) toward a flag pole. Below is that very such scene.

Write, in a file called `mario.c`, a program that recreates this half-pyramid using asterisks (`*`) for blocks. However, to make things more interesting, first prompt the user for the half-pyramid's height. (The height of the half-pyramid pictured above happens to be 8.) Then, generate the desired half-pyramid. Assume again that the user's terminal window is exactly 80 characters wide by 24 characters tall. So that a blinking prompt still fits on the screen after a half-pyramid's generation, demand that the user provide a non-negative integer no greater than 23. Take care to align your half-pyramid, no matter its height, in the window's bottom-right corner (*i.e.*, 80 characters over and 23 characters down). Note that the rightmost two columns of blocks are of the same height. No need generate the pipe, clouds, or Mario himself, though we'll be impressed if you come up with a neat way.

6. (15 points.) Read up on C's "bitwise operators." Then write, in `binary.c`, a program that converts decimal numbers to binary. This program must first prompt the user for a non-negative, base-10 integer. It must then print the base-2 equivalent as a sequence of 32 characters (namely '0' and '1'), from highest-order bit to lowest-. Your program must require that the user's input be between 0 and $2^{31} - 1$. (The first character you print will thus always be '0'.) See in just how few lines you can implement this program.

7. That's it for this problem set! No free point this time! I know, even though you did the Hacker Edition.

Submitting Your Work.

8. Ensure that your work is in `~/cs50/ps1/`. Submit your work by executing the command below.

   ```
   cs50submit ps1 hacker
   ```

   Thereafter, follow any on-screen instructions until you receive visual confirmation of your work's successful submission. You will also receive a "receipt" via email to your FAS account, which you should retain until term's end. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.