Laboratory Exercise 2

cpe 357 Fall 2010

Due by (or before) 4:00pm, Wednesday, October 6th[1].
The Written Exercises (problems) are to be done individually.
The Laboratory Exercises are also to be done individually.

This lab is a little longer than usual because it spans a little longer than a week. None of the tasks are terribly large.

Problems

These problems are designed to provoke some thought:

1. (Warm up) Please provide declarations for the following data:
   (a) a pointer *cp* that points to a `char`.
   (b) a pointer *ap* that points to an array of `char`s.
   (c) a pointer *pp* that points to a pointer that points to an `int`.
2. Is it possible in C to declare and initialize a pointer that points to itself? Why or why not? (And, if so, *how,* of course.)
3. What is the fundamental problem with the following code fragment intended to print out a string:

   ```
   char s[] = "Hello, world!\n";
   char *p;
   for(p = s; p != '\0'; p++)
     putchar(*p);
   ```

   What will happen when this is executed? How can it be fixed?
4. C programmers often say "arrays are the same as pointers". In one sense this is true. In another, more correct, sense they are fundamentally different.
   (a) In what ways is this statement correct?
   (b) How is it in error? That is, what makes a pointer fundamentally different from an array?
5. List the three flaws you discover in part 1 of the laboratory exercise. (See below.)
6. Exercise 1.3 from Stevens APUE.
7. Exercise 1.5 from Stevens APUE.
8. A subcase of Ex. 2.2 from Stevens: On `vogon`, what is the actual data type of a `size_t` (the type of `malloc()`'s argument)? In what header file is it defined?

[1] But it'd probably be useful to do it sooner rather than later.

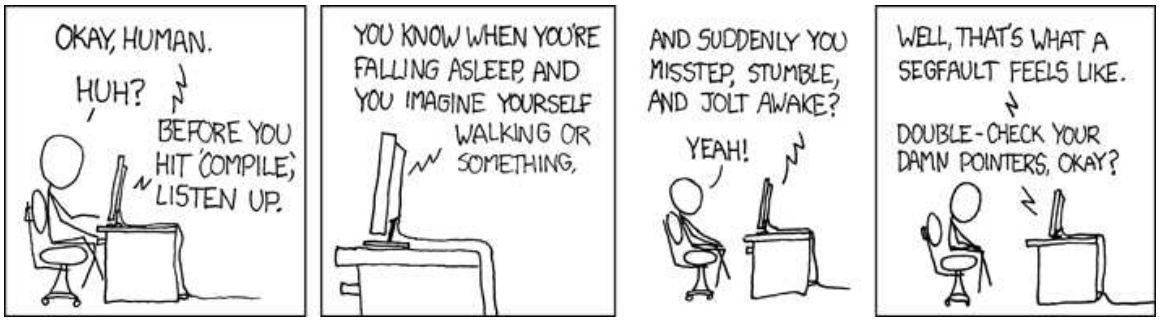

Figure 1: `http://xkcd.com/371/`

Laboratory Exercises

This hodgepodge of laboratory exercises designed to provide you with useful tools.

1. GDB Tutorial.

   Read and understand the *Quick Introduction to GDB* linked from the main cpe357 web page.

   Once you have done this, write a small program, compile it, and run it within gdb. At the very least, you should

   (a) set a breakpoint by function name,

   (b) print the value of some variable in the function,

   (c) examine a backtrace to see how the function was called, and

   (d) step through a loop.

   Once you have done all those things, download a copy of `labprog.c` and figure out what's wrong with it. It is available at:
   `http://www.csc.calpoly.edu/~pnico/class/f10/cpe357/lab/lab02/labprog.c`

   There are 3 significant flaws in it I expect you to note[2].

   You could probably figure these out by hand, but use the debugger. It's an important tool.

2. Learn to use `make(1)`.

   `make(1)` is a wonderful build-management tool that will help you keep your programs up to date while minimizing compile time by only recompiling those components for which it is necessary[3]. Besides, there is a direct benefit to this because every assignment from here on out will require you to submit a functional Makefile.

   Your task:

   (a) Read the *Quick notes on make* from the cpe357 web page.

   (b) Read and study the sample makefiles published with the solutions to Asgn1, also linked from the cpe357 page.

[2] There are other serious problems with this program, too, such as not checking the return value of `malloc()`, but that's not the sort of thing you're looking for.

[3] That is, it will do that if you set up your dependencies correctly. If you get those wrong, you can be in for a world of hurt.

(c) Write a small program consisting of at least two separate source files and create a `Makefile` for it. This makefile must compile the program when no argument is given to make, and support the following targets:

all	Build the program.
prog	(where *prog* is whatever your program is called) Build the program.
test	Builds the program and runs it with a sample input. (For example, if the program reads a string and reverses it, make test could execute a command like "`echo "hello" | myprog`")
clean	Removes all non-essential files generated during the build. Generally this means the intermediate object (.o) files.

Your goals when writing this makefile should be to minimize the amount of duplication. That is, `all`, `prog`, and `test` should not each have separate instructions for building the program. You should also make sure to include appropriate dependency information so that if one of your source files changes `make` will only recompile those files that need to be recompiled.

When finished: Write me a few sentences about the approach you took, problems you encountered, and lessons you learned and attach them, along with a printout of your makefile, to your written exercises above.

Bonus: Now—and only now—look into the man page for `makedepend(1)`, but be warned: `makedepend` will include all the system files, too, making makefiles that are not portable. Clever application of the `-Y` option can get around this.

3. Program: `uniq`

   This program is an exercise with dynamic data structures as a warm-up for Assignment 2.

   Write a version of the unix utility program `uniq(1)`. This program will act as a filter, removing adjacent duplicate lines as it copies its stdin to its stdout. That is, any line that is identical to the previous line will be discarded rather than copied to stdout.

   Your program may not impose any limits on file size or line length.

   To get started, I highly recommend writing a function `char *read_long_line(FILE *file)` that will read an arbitrarily long line from the given file into newly-allocated space. Once you have that, the program is easy.

   Be careful to free memory once you are done with it. A memory leak could be a real problem for a program like this.

Tricks and Tools

There are some library routines with which you might want to be familiar before attacking `labprog.c` listed in Figure 2.

`strlen(3)`	calculate the length of a string
`strcmp(3)`	compare two strings
`fgets(3)`	read a string into a buffer
`malloc(3)`	allocate a given number of bytes of memory from the heap
`realloc(3)`	Change the size of a previously allocated chunk of memory
`free(3)`	return a malloc()ed region of memory to the heap

Figure 2: Some potentially useful commands and library functions

What to turn in

For the Written Problems: individually written solutions to the problems according to the guidelines set forth in the syllabus.

For the Laboratory Exercise: Submit via `handin` to the `lab02` directory of the `pn-cs357` account:

- your well-documented source file(s) for `uniq`, and
- A makefile (called `Makefile`) that will build `uniq` from your source when invoked with no target or with the target "`uniq`".