Laboratory Exercise 2

cpe 357 Fall 2020

Due by 11:59pm, Monday, September 28th The Written Exercises (problems) are to be done individually.
The Laboratory Exercises are also to be done individually.

This looks kind of long, but none of the tasks below are terribly large.

Problems

These problems are designed to provoke some thought (could there be a quiz coming?):

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. Exercise 1.3 from Stevens APUE.
6. Exercise 1.4 from Stevens APUE, 3rd ed. (Ex. 1.5 in the 2nd.)
7. A subcase of Ex. 2.2 from Stevens: On `unix5`, what is the actual data type of a `size_t` (the type of `malloc()`'s argument)? In what header file is it defined?

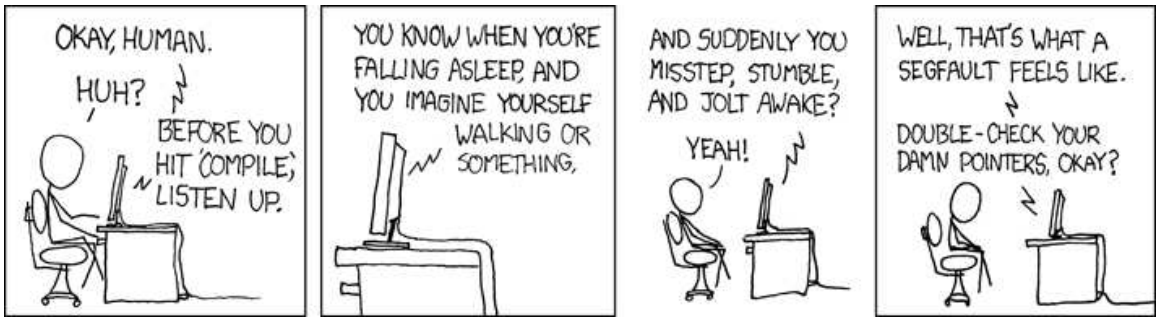

Figure 1: `http://xkcd.com/371/`

Laboratory Exercises

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

1. To help me with sorting out who's who in the class (and three years from now when you write me out of the blue asking for a recommendation) I am asking you to submit a digital photo of yourself to the `lab02` directory of the `pn-cs357` account. I will not be sharing these with anybody so it doesn't have to be glamorous, but it does need to be recognizable. That is, that great photo from last Halloween of you in your mummy costume really isn't the thing for this lab.

   Please make the root of the file name your login name. E.g., mine would be `pnico.jpg`.

2. 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.

   You might want to try this on the program from step 4.

3. 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[1]. 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.

[1] 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.

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

(c) Write a small program 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.

You might want to do this for the program from step 4.

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.

4. 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 working on `uniq` listed in Figure 2.

What to turn in

For the Written Problems: Include then in the a README file described below.

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

- your picture, named after yourself,

`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

- your well-documented source file(s) for `uniq`, and
- A makefile (called `Makefile`) that will build `uniq` from your source when invoked either with no target or with the target "`uniq`".
- A README file that contains:
  - Your name(s). In addition to your names, please include your Cal Poly login names with it, in parentheses. E.g. `(pnico)`
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.
  - The answers to the written questions above.

  The README file should be **plain text,** i.e, **not a Word document**, and should be named "README", all capitals with no extension.