Perilous Pointers

Entire Assignment due **2019-01-30 23:59**
Graded files:

- part1-functions.c
- part2-main.c

Content

Learning Objectives

The learning objectives for Perilous Pointers are:

- Pointers
- Strings
- Functions
- Function Pointers

Introduction

In CS 125, CS 225, and other classes, you have used various languages that are considered to be “C-based”, but up to now you may have very limited experience in C programming. This lab will provide a short programming introduction to pointers, strings, and functions in C.

This lab will be divided up into two parts. In the first part, you will be debugging broken functions that use pointers incorrectly. In the second part, you will need to write code to call some “creatively defined” functions so that each prints out “Illinois”.

For this lab, you should modify only:

- `part1-functions.c`
- `part2-main.c`

All other files will be replaced with new/different files for grading. If you modify any other files for debugging purposes, please ensure you test your program with the original files.

Part 1

There are erroneous/unimplemented functions in `part1-functions.c` . Your task is to modify the functions according to the comment above each function so that the output of `./part1` looks exactly as follows:

```
== one() ==
20 not passed!
100.000000 passed!
== two() ==
The value of p is: 4
== three() ==
x and y are equal.
x and y are different.
== four() ==
4 == 4.000000
432 == 432.000000
== five() ==
a is a letter.
a is not a letter.
== six() ==
Hello World!
== seven() ==
0.000000 0.100000 0.200000 0.300000 0.400000 0.500000 0.600000 0.700000 0.800000 0.900000
== eight() ==
0 10 40 90 160 250 360 490 640 810
== nine() ==
orange and blue!
ORANGE and blue!
Orange and BLUE!
orange and blue!
== ten() ==
The radius of the circle is: 17.500000.
The radius of the circle is: 10.000000.
== clear_bits() ==
170
0
171
0
20
0
== little finite automatons
5
4
6
7
```

Note that you can just diff with `part1-expected-output`.

Part 2

We have given you a file called `part2-functions.c`, that you may not change. Inside `part2-functions.c`, you will see twelve different functions, such as `first_step()`:

```
void first_step(int value) {
  if (value == 81)
    printf("1: Illinois\n");
}
```

To complete Part 2, you must write `part2-main.c` so that it makes calls to all eleven functions in `part2-functions.c`, and such that each one prints out its "Illinois" line. When running `./part2`, your output should look exactly like this:

```
 1: Illinois
 2: Illinois
 3: Illinois
 4: Illinois
 5: Illinois
 6: Illinois
 7: Illinois
 8: Illinois
 9: Illinois
10: Illinois
11: Illinois
```

Note that you can just diff with `part2-expected-output`.

You should **not** edit the part2-functions.c file. In fact, when we grade your program, we will replace the part2-functions.c file with a new version of the file (and we'll change the "Illinois" string so printing out "Illinois" in a for-loop will get you no credit).

Compile and Run

To compile the release version of the code, run:

```
make clean
make
```

This will compile your code with some optimizations enabled, and will not include debugging information. If you use a debugger on the 'release' build, it will not be able to show you the original source code, or line numbers. Optimizations sometimes expose bugs in your code that would not show up otherwise, but since optimizations tend to reorder your code while compiling, an optimized version of your code is not optimal for debugging.

You probably don't need to worry about the different build types very much for this assignment, but the distinction will become more important on future assignments.

To compile your code in debug mode, run `make debug` instead of `make` (`https://linux.die.net/man/3/make`)

To run Part 1:

```
./part1
```

or

```
./part1-debug
```

To run Part 2:

```
./part2
```

or

```
./part2-debug
```