Homework #1 – From C to Binary

Due date: see course website

Directions:

- For short-answer questions, submit your answers in PDF format as a file called <NetID>-hw1.pdf. ***Word documents will not be accepted.***
- For programming questions, submit your source file using the filename specified in the question.
- **You must do all work individually, and you must submit your work electronically via Sakai.**
    - All submitted code will be tested for suspicious similarities to other code, and the test will uncover cheating, even if it is "hidden" (by reordering code, by renaming variables, etc.).

Q1. Representing Datatypes in Binary

(a) [5 points] Convert $+47_{10}$ to 8-bit 2s complement integer representation in binary and hexadecimal.

(b) [5] Convert -13_{10} to 8-bit 2s complement integer representation in binary and hexadecimal.

(c) [5] Convert $+47.0_{10}$ to 32-bit IEEE floating point representation in binary and hexadecimal.

(d) [5] Convert -0.375_{10} to 32-bit IEEE floating point representation in binary and hexadecimal.

(e) [5] Represent the ASCII string "String for 250!" (not including the quotes) in hexadecimal.

(f) [5] Give an example of a number that cannot be represented as a 32-bit signed integer.

Q2. Memory as an Array of Bytes

Use the following C code for the next few questions.

```
float* e_ptr;

float foo(float* x, float *y, float* z){
    if (*x > *y + *z){
        return *x;
    } else {
        return *y+*z;
    }
}

int main() {
    float a = 1.2;
    e_ptr = &a;
    float* b_ptr = malloc (2*sizeof(float));
    b_ptr[0] = 7.0;
    b_ptr[1] = 4.0;
    float c = foo(e_ptr, b_ptr, b_ptr+1);
    free(b_ptr);
    if (c > 10.5){
        return 0;
    } else {
        return 1;
    }
}
```

(a) [5] Where do each of the following variables live (global data, stack, or heap)?

a. a
b. b_ptr
c. *b_ptr
d. e_ptr
e. *e_ptr

(b) [5] What is the value returned by main()?

Q3: Compiling and Testing C Code

[10] A high level program can be translated by a compiler (and assembled) into any number of different, but functionally equivalent, machine language programs. (A simplistic and not particularly insightful example of this is that we can take the high level code C=A+B and represent it with either add C, A, B or add C, B, A.)

When you compile a program, you can tell the compiler how much effort it should put into trying to create code that will run faster. If you type **`g++ -O0 -o myProgramUnopt prog.c`**, you'll get unoptimized code. If you type **`g++ -O3 -o myProgramOpt prog.c`**, you'll get highly optimized code.

Please perform this experiment on the program prog.c, linked on the course website. Compile it both with and without optimizations. **Compare the runtimes of each and write what you observe.** (To time a program on a Unix machine, type "**`time ./myProgram`**", and then look at the number next to the word "user". This number represents the time spent executing user code.)

Q4: Writing and Compiling C Code

In the next three problems, you'll be writing C code. You will need to learn how to write C code that:

- Reads in a command line argument (in this case, that argument is a filename such as "pizzainfo.txt"),
- Opens a file, and
- Reads lines from a file

You may want to consult the internet for help on this. You can find many examples for both fgets-based IO and fscanf-based IO, either of which can be made to work for these problems.

While you can consult resources to learn *how* a function works, you may not *use* any code from any external source (internet, textbook, etc.). Plagiarism of code will be treated as academic misconduct.

Your programs must run correctly on Duke Linux machines (login.oit.duke.edu). If your program name is myprog.c, then we should be able to compile it with: **`g++ -g –o myprog myprog.c`**

If your program compiles and runs correctly on some other machine but not on Duke Linux machines, the TA grading it will assume it is broken and deduct points. It is your job to make sure that it compiles and runs on Duke Linux machines. Code that does not compile or that immediately fails (e.g., with a segmentation fault) will receive approximately zero points – it is NOT the job of the grader to figure out how to get your code to compile or run.

All files uploaded to Sakai should adhere to the naming scheme in each problem and must match the case shown. If file names do not adhere, they will not be seen by the auto-grader and may receive a score of 0.

All programs should print their answers to the terminal in the format shown in each problem. If not adhered to, the problem may receive a score of 0.

About the self-tester and auto-grader

These questions will provide you with a self-test tool, and the graders will be using a similar tool (but with more test cases) to conduct grading. If you encounter issues or have questions, please post on the Piazza so we can address them.

A suite of simple test cases will be given for each problem, and a program will be supplied to automate these tests on the command line. The test cases can *begin* to help you determine if your program is correct. However they will *not be comprehensive*, it is up to you to create test cases beyond those given to ensure that your program is correct.

Test cases will be supplied in the starter kit linked on the course website. Download this file to your working directory and extract its contents typing the command into your terminal:

```
wget http://people.duke.edu/~tkb13/courses/ece250/homeworks/homework1-kit.tgz
tar -xvzf homework1-kit.tgz
```

Within these files there is a program that can be used to test your programs. It can be run by typing:

```
./hw1test.py
```

If run without arguments, as above, the tool will print a help message:

```
Auto Tester for Duke CS/ECE 250, Homework 1, Fall 2019

Usage:
  ./hw1test.py <suite>

Where <suite> is one of:
  ALL            : Run all program tests
  CLEAN          : Remove all the test output produced by this tool in
                   tests/
  byseven        : Run tests for byseven
  recurse        : Run tests for recurse
  PizzaCalc      : Run tests for PizzaCalc
```

To properly use the test program you must first compile it. You should name your executable after the .c file. For instance, problem (a)'s source code should be called byseven.c, and the executable called byseven. To compile, you would use the command:

```
g++ -g -o byseven byseven.c
```

Once your code compiles cleanly (without compiler errors), the tests can be run.

The tester will output “pass” or “fail” for each test that is run. If your code fails a particular test, you can run that test on your own to see specific errors. To do this, run your executable and save the output to a file. Shown next is an example from problem (a). After compiling, pass your program a parameter from one of the tests (listed in the tables below) and redirect the output to a file (output will also print to the screen):

```
./byseven 2 |& tee test.txt
```

Here, 2 is the parameter. The “`|& tee test.txt`” part tells your output to print to the screen and to a file called “test.txt”. (See here for more about I/O redirection.)

If you see no errors during runtime, compare your program's output to the expected output from that test as seen in the table using the following command:

```
diff test.txt tests/byseven_expected_1.txt
```

If nothing is returned your output matches the correct output, if diff prints to the screen then you are able to see what the difference between the two files is and what is logically wrong with your program. (See here for an introduction to diff.)

We used “**`byseven_expected_1.txt`**” above, because test ID 1 is the test that has an input of “2”; you can see what each test does by consulting the test tables for each program below.

Alternately, you may review the actual output and diff against expected output that are automatically produced by the tool. The files the tool uses are:

- Input data is stored in: `tests/<suite>_input_<test#>.txt`
- Expected output is stored in: `tests/<suite>_expected_<test#>.txt`
- Actual output is logged by the tool in: `tests/<suite>_actual_<test#>.txt`
- Diff output is logged by the tool in: `tests/<suite>_diff_<test#>.txt`

Q4a: byseven.c

[10] Write a C program called byseven.c that prints out the first N positive numbers that are divisible by 7, where N is an integer that is input to the program on the command line. Since your binary executable is called byseven, then you'd run it on an input of 4 with: **`./byseven 4`**. Your output in this case should look like:

```
7
14
21
28
```

Be sure that your main function returns EXIT_SUCCESS (0) on a successful run. **(-25% penalty per test with a non-zero exit status!)**

You will upload byseven.c into Sakai (via Assignments).

The following are the tests done within the auto test program for this problem:

Test Number	Parameter Passed	What is Tested
0	1	input of 1
1	2	input of 2
2	3	input of 3
3	4	input of 4

Q4b: recurse.c

[20] Write a C program called recurse.c that computes f(N), where N is an integer greater than zero that is input to the program on the command line. f(N) = 3*(N-1)+f(N-1)+1. The base case is f(0)=2. Your code must be recursive. **The key aspect of this program is to teach you how to use recursion; code that is not recursive will be severely penalized! (-75% penalty!)**

Your program should output a single integer.

Be sure that your main function returns EXIT_SUCCESS (0) on a successful run. **(-25% penalty per test with a non-zero exit status!)**

You will upload recurse.c into Sakai.

The following are the tests done within the auto test program for this problem:

Test Number	Parameter Passed	What is Tested
0	1	input of 1
1	2	recursion of 2
2	3	deeper recursion
3	4	even more recursion

Q4c: PizzaCalc.c

[50] Write a C program called PizzaCalc to identify the most cost-effective pizza one can order[1]. The tool will take a file as an input (eg., "**`./PizzaCalc pizzainfo.txt`**"). The format of this file is as follows. The file is a series of pizza stats, where each entry is 3 lines long. The first line is a name to identify the pizza (a string with no spaces), the second line is the diameter of the pizza in inches (a float), and the third line is the cost of the pizza in dollars (another float). After the last pizza in the list, the last line of the file is the string "DONE". For example:

```
DominosLarge
14
7.99
DominosMedium
12
6.99
DONE
```

Your program should output a number of lines equal to the number of pizzas, and each line is the pizza's name and pizza-per-dollar (in^2/USD). The lines should be sorted in *descending* order of pizza-per-dollar, and you must write your own sorting function (you can't just use the qsort library function). Pizzas with equal pizza-per-dollar should be sorted alphabetically (e.g. based on the strcmp function). For example:

```
DominosLarge 19.26633793
DominosMedium 16.17987633
BobsPizza 11.2
JimsPizza 11.2
```

To refresh your memory, the formula for the area of a circle is πr^2, and to compute area based on diameter, $\frac{\pi}{4}d^2$. Then just divide by cost to get pizza-per-dollar. You should compute using 32-bit `float`s and define π with:

```
#define PI 3.14159265358979323846
```

You may assume that pizza names will be less than 64 characters.

To mitigate the divide-by-zero situation, if the cost of the pizza is zero, the pizza-per-dollar should simply be zero (as the free pizza must be some kind of trap). A pizza with diameter zero will naturally also have a pizza-per-dollar of zero, as mathematical points are not edible. If your program is fed an empty file the program should print the following and exit:

```
PIZZA FILE IS EMPTY
```

[1] This program only optimizes single pizza prices, but of course most pizza deals involve getting multiple pizzas; we'll ignore this fact. We also are ignoring toppings, pizza thickness, quality, etc. You are welcome to enhance your pizza calculator further outside of class.

Files of the wrong format will not be fed to your program. In all cases, your program should exit with status 0 (i.e., main should return 0).

Important notes:

- **You will need to use dynamic allocation/deallocation of memory, and points will be deducted for improper memory management (e.g., never deallocating any memory that you allocated). The test script checks for this, but to actually diagnose memory leaks, you use valgrind with the `--leak-check=yes` option. (-50% penalty per test with a memory leak!)**
- **You may NOT read in the input file more than once. This means you cannot count the number of entries ahead of time -- you will instead need to allocate memory *dynamically* over the course of the run. Many programming problems tasks involve input of unknown size that you cannot simply read twice; dynamic memory management is therefore essential. (-50% penalty overall!)**
- **Internally, C's fopen() call malloc's space to keep track of the open file. Therefore, to avoid a memory leak (and the accompanying penalty), you must fclose() the opened file before exiting.**
- Be sure that your main function returns EXIT_SUCCESS (0) on a successful run. **(-25% penalty per test with a non-zero exit status!)**
- The self-tester, when looking at floats, checks to see they're within 0.1%, so you don't have to worry if you're off by a tiny amount from the published outputs due to floating point error.

You will upload PizzaCalc.c into Sakai.

The following are the tests done within the auto test program for this problem:

Test #	Parameter Passed	What is Tested
0	tests/PizzaCalc_input_0.txt	One pizza
1	tests/PizzaCalc_input_1.txt	Two pizzas, in order
2	tests/PizzaCalc_input_2.txt	Two pizzas, out of order
3	tests/PizzaCalc_input_3.txt	Six pizzas
4	tests/PizzaCalc_input_4.txt	Ensure we stop reading at "DONE"
5	tests/PizzaCalc_input_5.txt	Correct output with diameter of zero
6	tests/PizzaCalc_input_6.txt	Correct output with cost of zero
7	tests/PizzaCalc_input_7.txt	100 pizzas, some stats are zero
8	tests/PizzaCalc_input_8.txt	Empty file