Homework #1 – From C to Binary
Due Fri, Jan 28 at 5:00pm Eastern (in Gradescope)
130 points total

You must do all work individually. You may not look at each others' code, you may not help each other debug, etc. Invariably some students think that they will outsmart the tool for detecting cheating, and they end up receiving a zero on the entire assignment and referral to the Office of Student Conduct; please do not do that.

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.).

You will be graded strictly on what you submit and when you submit it. Please double-check what you have uploaded to Gradescope.

IMPORTANT NOTE 1: For programming assignments, Gradescope allows you to "activate" older submissions, but we will grade your latest submission. You may not make a submission after the deadline and then activate an older, on-time, submission if your late submission does not improve your score. You've been provided with a (non-comprehensive) test kit which you must use to test your code locally before submitting to Gradescope. Gradescope is meant for submission and not testing. You will be penalized for late submissions even if they are identical to older, on-time submissions.

IMPORTANT NOTE 2: Lost or corrupted files are NOT a valid reason for an extension, so please back up your work frequently! The Unix container you use is on a file system that should be reliable, but you are still responsible for making additional backups. (This is an important issue beyond just this class – always back up your work frequently.) You can copy files from your Unix container (or your personal laptop or wherever) to Duke's Box storage at box.duke.edu. If you're comfortable with git, you can also backup to git, **as long as you ensure that your repository is private**.[1] You can use other backup services, too, but the responsibility is on you to backup your work.

Directions:

- For short-answer questions 1 through 9, submit your answers in a single PDF file called <NetID>-hw1.pdf. ***Word documents will not be accepted.***

[1] Please be extremely careful about repo privacy! If your repo is not private, it is accessible by others and is thus academic misconduct.

- For programming questions, submit your source file using the filename specified in the question.
- Download starter code from Sakai. Go to Assignments > Homework 1 and find starter code files in the same location as this document.
- **You must submit your work electronically via Gradescope.** Submission consists of 4 files: <NetID>-hw1.pdf, mersenne.c, recurse.c, covidtracker.c.

Representing Datatypes in Binary

1) [5 points] Convert $+49_{10}$ to 8-bit 2s complement integer representation. Show your work!
2) [5] Convert -37_{10} to 8-bit 2s complement integer representation. Show your work!
3) [5] Convert $+49.5_{10}$ to 32-bit IEEE floating point representation. Show your work!
4) [5] Convert -5.75_{10} to 32-bit IEEE floating point representation. Show your work!
5) [5] Represent the string "Banchero = 5" (not including the quotes) in ASCII. Use hexadecimal format for representing the ASCII. You must include the null character at the end of the string. (You don't need to show your work.)
6) [5] Give an example of a number that cannot be exactly represented by a 32-bit computer. Explain your answer.

Memory as an Array of Bytes

Use the following C code for the next few questions.

```
float* points_ptr;

int main() {
     float avg_per_game = 21.34;
     points_ptr = &avg_per_game;
     float* rebounds_ptr = (float*) malloc (2*sizeof(float));
     rebounds_ptr[0] = 7.0;
           *(rebounds_ptr+1) = 4.0;
     float num = foo(points_ptr, rebounds_ptr,
rebounds_ptr[1]);
     free rebounds_ptr;
     if (num > 10.5){
                return 0;
           } else {
               return 1;
           }
}

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

7) [5] Where do the following variables live (global data, stack, or heap)?
   a. avg_per_game
   b. rebounds_ptr
   c. *rebounds_ptr
   d. points_ptr
   e. rebounds_ptr[0]
8) [5] What is the value returned by main()? Explain your answer.

Compiling C Code

9) [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 located in the "Resources" section of the course Sakai site (in the folder named "Homework resources"). Compile it both with and without optimizations. **Compare the runtimes of each (how long the program takes to run, not how long it takes to compile) and write what you observe and whether it is what you expected or not.** (To time a program on a Unix machine, type "time ./myProgram", and then look at the number before the "u". This number represents the time spent executing user code.)

Writing 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 "statsfile.txt" without the quotes),
- Opens a file, and
- Reads lines from a file

You may want to consult Recitation #2 and/or the internet for help on this. One (of many!) examples of helpful information is at http://www.phanderson.com/files/file_read.html.

NOTE: We are not trying to trick you. You will always get valid inputs (i.e., you don't have to check that an input is valid). E.g., if we say the input will be an integer, we won't try to break your code by giving it a non-integer input.

Rules

You may not use/borrow any code from any external source (internet, textbook, etc.). Plagiarism of code will be treated as academic misconduct.

Your programs must run correctly on the virtual machine provided by your container (i.e., the virtual machine you learned about in Recitation #1). If your program name is myprog.c, then we should be able to compile it with: g++ –o myprog myprog.c.

If your program compiles and runs correctly on some other machine but not on container machine, 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 container machine. Code that does not compile or that immediately fails (e.g., with a seg 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 Gradescope 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.

Testing and Grading Your Code

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.

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.

Note: I'm not trying to trick you. I will not give you bizarre, tricky inputs. If the program takes an integer as an input, I'll give you an integer and you don't have to check that the input is an integer.

Test cases will be supplied in the starter kit available in the Resources section of Sakai (under "Homework" and called homework1-kit.tar.gz). Place this file in your Unix homework working directory,[2] and extract its contents by typing the following command into your Unix terminal:

```
tar -xvzf homework1-kit.tar.gz
```

This will create a directory called **kit**, and there will be files in this directory. Go into this directory (using cd) and move your code (your .c files) into this directory. Within the files that came with the kit, 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, Spring 2022

Usage:
  ./hw1test.py <suite>

Where <suite> is one of:
```

[2] Remember: you have a browser in your Unix machine. Use it to navigate to Sakai.

```
ALL             : Run all program tests
CLEAN           : Remove all the test output produced by this tool in
                  tests/
mersenne        : Run tests for mersenne
recurse         : Run tests for recurse
covidtracker    : Run tests for covidtracker
```

To properly use the test program you must first compile your code using what was learned in problem 3. You should name your executable after the .c file. For instance, problem (a)'s source code should be called mersenne.c, and the executable called mersenne. To compile, you would use the command:

```
g++ -g -o mersenne mersenne.c
```

There is also a Makefile in the kit directory, and you can make mersenne with this command:

```
make mersenne
```

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):

```
./mersenne 9 |& tee myTest2.txt
```

Here, 9 is the parameter. The "`|& tee myTest2.txt`" part tells your output to print to the screen and to a file called "myTest2.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 myTest2.txt tests/mersenne_expected_9.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.)

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`

The Coding Tasks

10) [10] Write a C program called mersenne.c that prints the n^{th} Mersenne number (any number that is 2^n-1), where n is the command line argument, and the first Mersenne number is 2^1- 1 = 0, the second Mersenne number is 2^2-1 = 3, etc. For example, to run it with an argument of 4, you'd do the following:

    ./mersenne 4

    For the input 4, your program should print the output 15.

    I guarantee that the input, n, will be an integer that is greater than 0. I also guarantee that the output will fit in a variable of type "int" (i.e., you don't need to use a "long").

    You will upload `mersenne.c` to Gradescope.

11) [20] Write a C program called recurse.c that computes and prints out f(n), where n is an integer greater than zero that is input to the program on the command line. f(n) = 3*n + [2*f(n-1)] - 2. 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, *even if it gets the right answer*, will be severely penalized!

    I guarantee that the input, n, will be an integer that is greater than or equal to 0. I also guarantee that the output will fit in a variable of type "int" (i.e., you don't need to use a "long").

    You will upload `recurse.c` to Gradescope.

12) [50] Write a C program called covidtracker.c that takes a file as an input (./covidtracker statsfile.txt). The file is in the following format. The file is a series of patients and who they infected, not including Patient Zero, the Blue Devil. To simplify life, each patient can infect no more than two other people. Each entry in the file is the patient's name first and then who infected him/her. The entries in the file are ordered by the time of infection, i.e., the n^{th} entry is n^{th} person to be infected. For example, in the example below, Jones was the 2^{nd} infection, and she was infected by Smith.

After the last patient in the list, the last line of the file is the string "DONE". For example:
statsfile.txt

Smith BlueDevil
Jones Smith
BuckyBadger Jones
Doe Smith
Johnson BlueDevil
White Doe
DONE

I guarantee that (a) no patient will appear in the list before the patient who infects him/her, and (b) no patient will be infected more than once, (c) no patients have the same name, and (d) every name is less than 30 characters.

Your program should output a list (data structure hint!) of all patients, including BlueDevil, that is ordered alphabetically. Each line of the output is the patient's name, followed by the names of the patient(s) they infected, if any, in alphabetical order. For the example input above, here is the output:

BlueDevil Johnson Smith
BuckyBadger
Doe White
Johnson
Jones BuckyBadger
Smith Doe Jones
White

Important Rules:

1) You may only read through statsfile once. Multiple passes are prohibited. You are not allowed to look through it to find out how many patients there are and then use that information to do static allocation of memory.
2) You must use dynamic allocation/deallocation of memory, and points will be deducted for improper memory management (e.g., never deallocating any memory that you allocated).
3) You must use malloc() for dynamic allocation of memory. It may be tempting to take advantage of the compiler's ability to understand an array declaration with a variable size (e.g., int myArray [NUMPATIENTS]), but we will deduct points for this. The point of this exercise is to teach you how to manage memory and use malloc().

Your program should return the value zero.
You will upload `covidtracker.c` to Gradescope.