Homework #3 – From C to Binary

Due date: see course website

Directions:

- This assignment is in two parts:
  - Questions 1-3 are written answer questions to be answered via PDF submitted to the GradeScope assignment “Homework 3 written”.
  - Question 4 consists of programming tasks to be submitted via GitLab transfer or upload to the GradeScope assignment “Homework 3 code”. Your source files must use the filenames specified in the question. **NOTE: Merely committing to GitLab is not sufficient!** You have to login to GradeScope and upload C files! Programs that show good faith effort will receive a minimum of 25% credit. (Refer to the rubric for more details.)
- **You must do all work individually, and you must submit your work electronically via GradeScope.**
  - 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 [30]

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

(b) [5] Convert -34_{10} to 8-bit 2s complement integer representation in binary and hexadecimal. You must show your work!

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

(d) [5] Convert -3.125_{10} to 32-bit IEEE floating point representation in binary and hexadecimal. You must show your work!

(e) [5] Represent the ASCII string “Duke 24!” (not including the quotes) in hexadecimal. You do not need to include the null character at the end.

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

Q2. Memory as an Array of Bytes [10]

Use the following C code for the next few questions.

```
int a = 42;

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

int main() {
    int* e_ptr = &a;
    int w = 2;
    int* b_ptr = (int*) malloc (2*sizeof(int));
    b_ptr[0] = 4;
    *(b_ptr+1) = 7;
    int c = foo(w, b_ptr, e_ptr);
    free(b_ptr);
    if (c > 10){
        return c;
    } 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]

[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 [80]

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 "golferstats.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 (either the Docker container or 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 and/or autograder will have to conclude that it is broken and deduct points. It is your job to make sure that it compiles and runs on Duke Linux machines. 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 not receive credit.

About the self-tester

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 Ed 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.

AGAIN: TESTING IS YOUR JOB (and could very well be your first job after graduating – THINK ABOUT TEST CASES THAT GO BEYOND THE ONES PROVIDED!

Test cases will be supplied in the repository that you will fork to begin the assignment. In our gitlab group for this semester on GitLab, find the repository "homework3". Fork the repository and clone it to your preferred environment to get started. (Review recitation 1 if you need a refresher.) Be sure the repo is marked private – not doing so is a violation of the Duke community standard!

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

```
./hwtest.py <test-suite>
```

Where <test-suite> is the name of one of the three programs you'll be writing, or the word "ALL" to run all the tests at once.

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

```
g++ -g -o sequence sequence.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):

```
./sequence 22 |& tee test.txt
```

Here, 22 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/sequence_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 "**`sequence_expected_1.txt`**" above, because test ID 1 is the test that has an input of "22"; 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`

You can tell exactly what the tool is doing by running it with the `-v` (verbose) flag – this will echo the commands executed so you can reproduce them yourself when needed:

About GradeScope and the auto-grader

When you upload your code to GradeScope, it will automatically kick off the auto-grader. (Hey, thanks for actually reading the assignment. Include a picture of a pitchfork in question 3 of your written submission PDF for one point of extra credit.) This program is very similar to the self-tester provided to you, but includes larger and more complex test cases whose result will *not* be shown to you until after the late deadline of the assignment. This mimics real life: you always test your software before releasing it, but your software's users will put it through inputs you may not have anticipated.

To submit via GradeScope, go to the "Homework 3 code" assignment, hit submit, and upload the relevant C files.

Q4a: sequence

[10] Write a **non-recursive** C program called sequence.c that prints out the Nth number of a sequence (specified below), where N is an integer that is input to the program on the command line. Each number in the sequence, S_N, is defined as $S_N = 3^N - 3$. The input value of N will be greater than or equal to zero. Since your binary executable is called sequence, then you'd run it on an input of 5 with: `./sequence 5`. Your output in this case should be 240.

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

To prepare you for the next assignment, where you will not have access to C libraries, you **MAY NOT** use the `math.h` library and its `pow` function to compute 3^N—you must compute it manually using repeated multiplication.

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	4	Input of 4
3	7	Input of 7
4	10	Input of 10

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) = 2*(N+1) + 3*[f(N-1)] - 17. The base case is f(0)=2. Your code must be recursive. **<u>The key aspect of this program is to teach you how to use recursion; code that is not recursive will be severely penalized!</u> (-75% penalty!)**

Your program should output a single integer.

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

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

Test Number	Parameter Passed	What is Tested
0	0	Base case
1	2	Just one level
2	4	Recursion
3	7	Deeper recursion

Q4c: golf.c

[50] Write a C program called golf.c to rank golfers based on their relationship to par. The program will take a file as an input (eg., "`./golf golferstats.txt`"). The format of this file is as follows. The file starts with an integer that is "par". The rest is a series of per-golfer stats, where each entry is 2 lines long. The first line is the golfer's last name, the second line is how many shots the golfer took. After the last golfer in the list, the last line of the file is the string "DONE". For example:

```
70
Woods
68
Wie
65
Scrub
73
DONE
```

Your program should output a number of lines equal to the number of players, and each line is the golfer's name and their relationship to par, which is computed as:

(shots taken by golfer) – (par).

If the result is positive, you must put a plus sign (+) before the metric. Note that 0 is neither positive nor negative.

The lines should be sorted in *ascending* order based on this metric, and you must write your own sorting function. Golfers with equal metrics should be sorted alphabetically (e.g., based on the strcmp function). For example:

```
Wie -5
Woods -2
Scrub +3
```

You may assume that golfer names will be fewer than 63 characters and that all numbers will fit into an int.

Empty files and 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).

<u>**Important notes:**</u>

- **You MUST use a linked list.**
- **You absolutely MUST use a linked list.**
- **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!)**

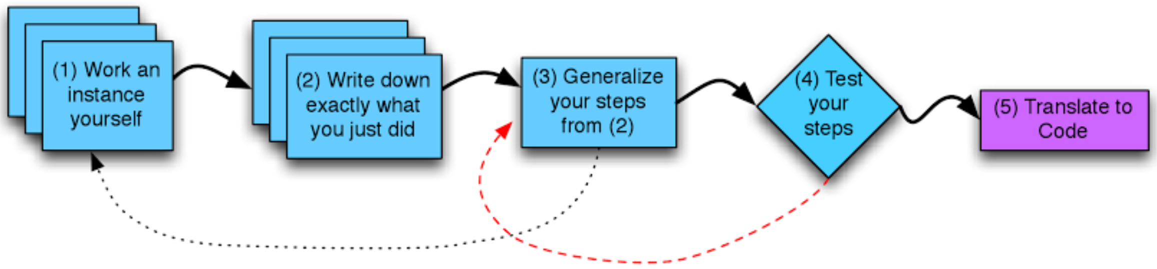
- **You may NOT read in the input file more than once. This means you cannot count the number of entries ahead of time -- you will instead need to allocate memory *dynamically* over the course of the run. Many programming problems tasks involve input of unknown size that you cannot simply read twice; 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 (i.e. 0) on a successful run. **(-25% penalty per test with a non-zero exit status!)**
- A common mistake is to forget to initialize data, especially memory provided via malloc. For example, if you malloc a linked list node for a golfer and never set its next pointer, that pointer's value is NOT null by default, but rather random junk, which can cause intermittent crashes. Valgrind can help you catch such things.
- For sorting, we strongly recommend using either insertion sort (i.e., inserting nodes into the linked list in the correct place) or bubble sort. If you use bubble sort, we recommend swapping values instead of pointers.
- You may find it helpful to write a function that compares two linked list nodes.

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

Test #	Parameter Passed	What is Tested
0	tests/golferstats_input_0.txt	One golfer
1	tests/golferstats_input_1.txt	Two golfers, in order
2	tests/golferstats_input_2.txt	Two golfers, out of order
3	tests/golferstats_input_3.txt	Six golfers
4	tests/golferstats_input_4.txt	Ensure we stop reading at "DONE"
5	tests/golferstats_input_5.txt	100 golfers

Appendix: How to Design

It's likely that you can slap together sequence and recurse without much thought, but golf will likely require you to *design* your program. Below is Prof. Hilton's recommended procedure for designing *any* software:



(1) Work an instance yourself
(2) Write down exactly what you just did
(3) Generalize your steps from (2)
(4) Test your steps
(5) Translate to Code

The "Hilton Method" for algorithm design

In (1), you would get a small input file and work it out yourself on paper. You might use cards or post-it notes to represent the records you're creating, and you'd move them about to simulate sorting. Write down how you did it, that's (2)!

In (3), you'd look at the steps you wrote out, and find what's common, and try to generalize it into a written algorithm. This is probably where you'd identify the data structures to use by asking, "what kind of movements do my procedures call for?"

In (4), to make sure you didn't screw up, you'd manually work some small cases by robotically following your written algorithm.

Only then, **when you have a written algorithm in hand**, should you start writing golf.c.

The most common advice I give in office hours is "figure out your algorithm on paper"!

Appendix: A word on data structures

A lot of students get tripped up on using data structures, especially since C is less helpful with error feedback. It is useful to remember (and perhaps even write down) the **invariant** of the data structure. The invariant is the set of rules that define it. For example, for a singly linked list, the invariant is "there's a head pointer that points to zero or mode nodes, linked via next pointers, til you get to NULL".

Then, in any operation that deals with that data structure, you start by assuming the invariant is true, then doing a series of steps that permutes the data structure such that the invariant remains true afterward.