Homework #1 – 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 1 written".
  - Question 4 consists of programming tasks to be submitted via GitLab transfer or upload to the GradeScope assignment "Homework 1 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.
- **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 -13_{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 -0.375_{10} to 32-bit IEEE floating point representation in binary and hexadecimal. You must show your work!

(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 [10]

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 = (float*) 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]

[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 **`gcc -O0 -o myProgramUnopt prog.c`**, you'll get unoptimized code. If you type **`gcc -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. **What is the runtime of each of the two versions of the program? What percent faster or slower is the optimized version?** (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.)

We'll revisit these programs in Homework 2 with an eye toward *how* optimization works.

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 "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 (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:
`gcc -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. 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 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 – 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 "homework1". 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 byseven.c, and the executable called byseven. To compile, you would use the command:

```
gcc -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
```

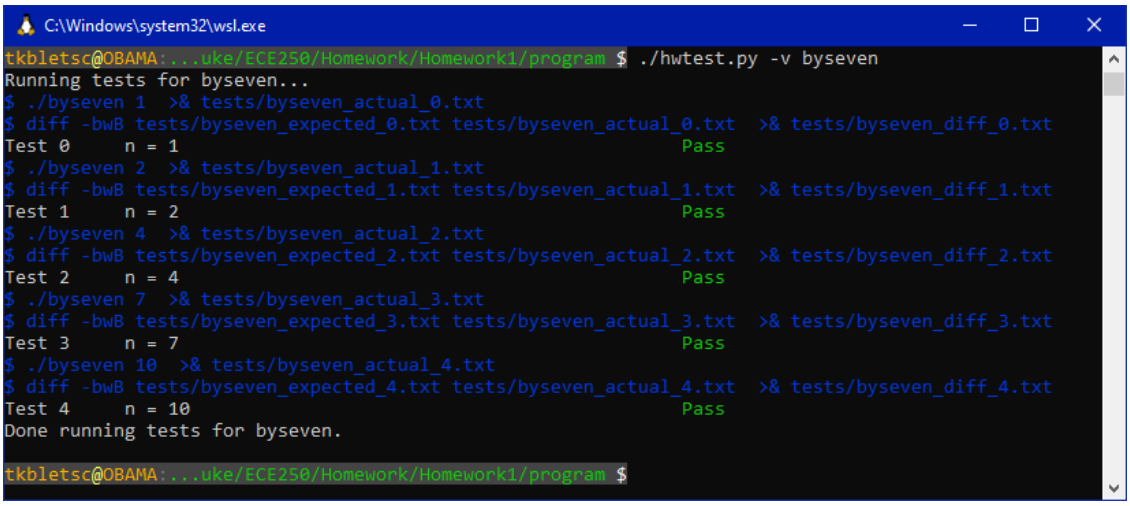
If nothing is returned your output matches the correct output, if diff prints to the screen then you are able to see what the difference between the two files is and what is logically wrong with your program. (See here for an introduction to diff.)

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`

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 fish 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 your, 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 1 code" assignment, hit submit, and upload the relevant C files.

Note for users of MacOS

MacOS does not provide **gdb**, but it does have **lldb**, which works similarly in most cases. Further, it does not have **valgrind**, nor is there an easy way to install valgrind ☹. Use of the container environment is recommended in this case.

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!)**

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 or equal to 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!)**

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: 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 fewer than 63 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:

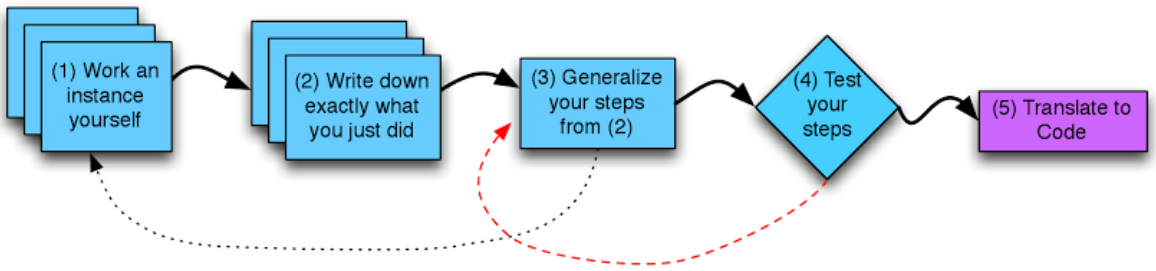
- **You will need to use dynamic allocation/deallocation of memory, and points will be deducted for improper memory management (e.g., never deallocating any memory that you allocated). The test script checks for this, but to actually diagnose memory leaks, you use valgrind with the `--leak-check=yes` 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.**
- **No AI stuff, please:** ChatGPT and friends might be able to write this. They might even do a decent job. But Homework 2 is to write the same programs in MIPS Assembly language, which they for sure can't do, and you won't be able to write it either if you don't understand your own code!
- 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.
- 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 pizza 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.

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

Appendix: How to Design

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



The "Hilton Method" for algorithm design

In (1), you would get a small pizza file and work it out yourself on paper. You might use cards or post-it notes to represent the pizza 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 PizzaCalc.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.