# Homework #1 – From C to Binary
# Due Weds, Feb 1 at 5:00pm (in Sakai)
# 130 points total

**<u>You must do all work individually</u>. You may not look at each others' code, you may not help each other debug, etc.**

All submitted code will be tested for suspicious similarities to other code, and the test <u>will</u> uncover cheating, even if it is "hidden" (by reordering code, by renaming variables, etc.).

**<u>Directions:</u>**

- For short-answer questions 1 through 9, submit your answers in a single PDF file called <NetID>-hw1.pdf. ***<u>Word documents will not be accepted.</u>***
- For programming questions, submit your source file using the filename specified in the question.
- **You must submit your work electronically via assignments section of Sakai.** Submission contents 4 files: <NetID>-hw1.pdf, fibtimes2.c, recurse.c, HoopStats.c.

## Representing Datatypes in Binary

1) [5 points] Convert $+39_{10}$ to 8-bit 2s complement integer representation. Show your work!
2) [5] Convert $-27_{10}$ to 8-bit 2s complement integer representation. Show your work!
3) [5] Convert $+39.0_{10}$ to 32-bit IEEE floating point representation. Show your work!
4) [5] Convert $-1.125_{10}$ to 32-bit IEEE floating point representation. Show your work!
5) [5] Represent the string "2017: K>Roy" (not including the quotes) in ASCII. Use hexadecimal format for representing the ASCII. (Don't need to show work.)
6) [5] Give an example of a number that cannot be 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* e_ptr;

int main() {
     float a = 21.34;
     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;
     }
}

float foo(float* x, float *y, 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. a
   b. b_ptr
   c. *b_ptr
   d. *e_ptr
   e. b_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 and write what you observe.** (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 the internet for help on this. One (of many!) examples of helpful information is at http://www.phanderson.com/files/file_read.html.

### 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 Linux machines in the login.oit.duke.edu cluster. 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 the Linux machines in the login.oit.duke.edu cluster, 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 the login.oit.duke.edu machines. 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 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.

## 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. These tools are relatively new, and we ask for your feedback and understanding as we work to develop 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 insure 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 linux homework working directory (you may find the CIFS ability to mount your OIT filesystem on your local machine useful for this), and extract its contents by typing the following command into your linux 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 2017

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/
  fibtimes2      : Run tests for fibtimes2
  recurse        : Run tests for recurse
```

```
    HoopStats       : Run tests for HoopStats
```

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 fibtimes2.c, and the executable called fibtimes2. To compile, you would use the command:

```
g++ -g -o fibtimes2 fibtimes2.c
```

There is also a Makefile in the kit directory, and you can make fibtimes2 with this command:

```
make fibtimes2
```

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):

```
./fibtimes2 2 |& tee myTest2.txt
```

Here, 2 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/fibtimes2_expected_2.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:

| | |
|---|---|
| • HoopStats 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 fibtimes2.c that prints out the first N fibonacci numbers times two, each one on its own line, where N is an integer that is input to the program on the

command line. (The first Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, etc. The first numbers your program should produce are 2, 2, 4, 6, 10, 16, 26, etc.) If your binary executable is called fibtimes2, then you'd run it on an input of 21 with: ./fibtimes2 21

You will upload fibtimes2.c into Sakai (via Assignments).

11) [20] Write a C program called recurse.c that computes and prints out f(N), where N is an integer <u>greater than zero</u> that is input to the program on the command line. f(N) = 3*(N-1)+[2*f(N-1)]-1. The base case is f(0)=5. **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, *even if it gets the right answer*, will be severely penalized!</u>**

    You will upload recurse.c into Sakai.

12) [50] Write a C program called HoopsStats that takes a file as an input (./HoopsStats statsfile.txt). The file is in the following format. The file is a series of player stats, where each player entry is 3 lines long. The first line is the player's last name, the second line is his jersey number (an int), and the third line is his year of graduation (an int). After the last player in the list, the last line of the file is the string "DONE". For example:

    statsfile.txt

```
Hill
33
1997
Battier
31
2001
Hurley
11
1996
DONE
```

    Your program should output a number of lines equal to the number of players, and each line is the player's name and his jersey number. The lines should be sorted in ascending order of graduation year, and you must write your own sorting function (you can't just use the qsort library function). In the case of equal graduation years, sort by last name. (I guarantee that this criterion will break all ties.) For example, the output for the sample statsfile above should be:

```
Hurley 11
Hill 33
```

Battier 31

**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).**

**Note also that 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 [NUMPLAYERS]), but we will deduct points for this.   The point of this exercise is to teach you how to manage memory and use malloc().**

You will upload HoopStats.c  into Sakai.