CISC 340 – HW 0
All programming languages are an interface to memory

Due date: Class time, Wednesday, February 26
(5 points)

Instructions:

All code must be developed on rusty. Submissions must be typed, preferably typeset in LaTeX. Submit your solution files and typeset PDF in a directory named "yourusername_hw0" in a single zip file via canvas and hand in a hardcopy of the PDF during class on the due date.
The pdf writeup of your homework must be included in the submission.

I guarantee all of the programming skills you will learn from this homework will prove useful.

1. Write a Makefile for a C program held entirely in the file myarr.c. Your Makefile should contain standard compiler and compiler flag variables, as well as standard rules for building/cleaning the project.

2. Here you will create a C program named myarr.c that is built using the Makefile from Part 1. This program will creates an integer array with 20 elements. The elements will be filled with the first 20 Fibonacci numbers (starting with 0 and 1) either via a function or a loop. You will not hard code the entries (other than the first two terms, 0 and 1). Once the array is filled, print out its contents. **Include a screenshot showing your code executing (fonts must be legible).**

3. In C, the unary & operator returns a memory address. For example, &i, would return the memory address of some variable i. If a variable contains a memory location it is a pointer. To access the value of the memory location that the pointer is "pointing to", it is necessary to "de-reference". De-referencing is done using the unary * operator. For example, the following code should print 10:

```
int my_int = 10;             // A regular int
int* my_intp = &my_int;      // my_intp has the type of int pointer,
                             // notice that * is used to define the pointer type here
printf("%i\n", *my_intp);    // * is used as an operator here to de-reference my_intp
```

In C, arrays can be thought of as pointers. Assuming A is an integer array, A[10] is functionally equivalent to *(A+10).

Extend your program from problem 2 to have a function that copies the contents of one array to another array using using memcpy(). This function should take two pointers and an integer (size of the arrays) as input. Your program should copy your first array into a new array named my_arr2. You will then overwrite the contents of my_arr2 with the first 20 square numbers (0, 1, 4, 9, etc.) and then print out the contents. The catch is that you cannot use the [] operator for array access. You **MUST** use pure pointer arithmetic.

You can create the array using:

```
int* myarr2 = (int*)malloc(20*sizeof(int));
// The sizeof() operator returns the size of a type/array/struct in bytes.
```

It is good practice to “free” this memory allocation once you are done with it. This level of control is taken away from you in Java as memory is freed automatically (and possibly at inconvenient times) by the garbage collector. After printing out the contents of myarr, make sure to call:
free(myarr);

Include a screenshot showing your code executing (fonts must be legible).

4. When using bash in Linux (your command line), you can use special characters to redirect the input/output from programs.
“./program > file” will overwrite the contents of file with the output from ./program.
“./program >> file” will append to file with the output from ./program.
“./program1 | ./program2” will send the output from program1 to the input of program 2. This special character is called a “pipe” as it provides a pipe for I/O between programs. You can find it with shift-backslash (just above the enter key).

Write a linux command line “one-liner” (a series of commands on 1 line strung together with >, >> , or |, only hitting enter once) to find the frequencies of all of the CPU cores on the departmental Linux machine and store the results into the file cpu_freqs.txt.
HINTS: The file /proc/cpuinfo, the commands cat, grep, and cut. Use google or the system manual pages to determine how these commands work.

When viewing the contents of your file I should see something like:

```
$cat cpu_freqs.txt
899.941
899.941
899.941
811.835
900.324
901.102
```

Now write a C program named mycat.c (and appropriate Makefile) to read in your cpu_freqs.txt file and print the contents (HINT: use fgets()).
You must use the gnu getopt library to get the name of the file (cpu_freqs.txt) from the command line.
You must also include functionality (with gnu getopt) to optionally get the name of an output file from the command line.

When you run your program on the command line (input file only), it should look like:

```
$./my_program -f cpu_freqs.txt
899.941
899.941
899.941
811.835
900.324
901.102
```

When you run your program on the command line (input and output file supplied), it should look like:

```
$./my_program -f cpu_freqs.txt -o output.txt
```

```
$
```

Include a screenshot showing your code executing (fonts must be legible).

5. Go to https://logic.ly/ and use the drag and drop interface to build a Boolean logic circuit for the following truth table:

A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Try to use the minimum amount of AND, OR, and NOT gates to create a correct circuit. Your circuit should be well organized with inputs on the left and outputs on the right. **Provide screenshots proving correct functionality.**

6. Extend your C program from part 4 to round the first four CPU frequencies to the nearest integer value. You will then store each of the rounded frequencies into a packed 64 bit unsigned integer (data type unsigned long long int). The format of this packing is:

Freq 3	Freq 2	Freq 1	Freq 0
2 B	2 B	2 B	2 B

You will need to use bitwise operations and make any constants 64 b as well (0xFFLLU would be a 64 b 0xFF). When executed, your program should display (your frequencies might vary):

```
$./my_program -f cpu_freqs.txt
899.941
899.941
899.941
811.835

Packed freqs: 228561117115319072
```

Include a screenshot showing your code executing (fonts must be legible).