This lab will be done with a partner.

1. Lab Overview and Goals:

For the first lab, you'll implement a program that sorts floating-point numbers using a sorting algorithm of your choice in C. Your program will read a collection of unsorted floats from a file, store those floats in an array, provide some information about the floats to the user, sort them from smallest magnitude to largest magnitude (i.e., ascending order), and print them out in sorted form to the user

1.1. Lab Goals:

Practice C programming basics: declaring variables, types, arrays, and functions. Write C code and functions that use statically declared arrays.

Gain experience writing and using C functions: function prototypes and definitions; pass by value; C array parameters.

C I/O: printf and file I/O using functions from our readfile library.

Apply top-down design to a C program.

Compile a C program with a Makefile.

2. Starting Point Code

2.1. Getting Your Lab Repo

Both you and your partner download the Lab1 files from Canvas.

2.2. Test sharing changes

As you and your partner work on your joint lab solution, you will want to share changes.

2.3. Starting Point files

Cloning the repository will give you the following starting point code files:

BASH

```
$ ls
Makefile                readfile.h   sorter.c
floats.txt              readfile.c
```

These files are:

1. `Makefile` A Makefile simplifies the process of compiling your program. We'll look at these in more detail later in the course. You're welcome to look over this one, but you shouldn't need to edit it for this lab. If you are interested, take a look at Section 8 for more information about make and makefiles.
2. `readfile.h` and `readfile.c` : These files contain a library for reading from files. You should make calls to functions in this library in your program to simplify file I/O. The instructions for using this library are explained below. Do not modify any code in these two files.
3. `sorter.c` : the file into which you will add your C solution and comments. The starting point code includes a helper function for you:

   `get_filename_from_cmdline` : takes the storage space for a string and the command line arguments, and initializes the string storage space with the filename from the command line argument. You should not change this function, but feel free to change where it gets called.
4. `floats.txt`: example input file to your program. This is provided for your convenience to help you test your program. You should create additional test files to more extensively test your program's correctness before you submit it for grading.

3. Compiling and Running

Run `make` to compiler your program. `make` uses rules in the Makefile to compile the binary executable, `sorter` , from `sorter.c` and from linking in the `readfile` library. Running `make` displays the `gcc` compiler output, which you should examine for errors and warnings. The starting point code has one warning telling you that the variable `values` is declared but not used in the program:

```
$ make
gcc -g -Wall -c readfile.c
gcc -g -Wall -o sorter sorter.c  readfile.o
sorter.c: In function 'main':
sorter.c:32:11: warning: unused variable 'values' [-Wunused-variable]
   32 |    float values[ARRAYSIZE];     /* stores values read in from file */
      |          ^~~~~~
```

To run the `sorter` executable binary, you need to include the name of the input file as a command line argument. The input file contains the set of floating point values for your program to sort, one value per line (see the `floats.txt` files included with the starting point). When run, your program will read in these values from the file (using the provided library), sort them, and

print out the sorted result back to the user. Here is an example of how to run the program:

```
$ ./sorter floats.txt
```
BASH

3.1. Sample Output

The following shows an example of what a run of a complete program looks like for a particular input file:

```
$ ./sorter floats.txt
Reading file named floats.txt
The file has 10 floats ranging from 0.00 to 9.00
The unsorted values are: 9.00 8.00 4.00 5.00 7.00 3.00 2 00 1.00 6.00 0.00
The sorted values are: 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00
```
BASH

4. Lab Details

Your complete program will:

1. read data from an input file into an array, using functions in the provided library
2. print out the unsorted values in the arrray
3. sort the array
4. print out the values in the sorted array

The main parts of the lab involve writing code using C arrays, functions, C command line arguments, and I/O.

4.1. File Format

The input file format consists of several lines of ASCII text. A properly formatted file will contain a short header, consisting of a single line with one integer and two floats on it. These numbers represent the total number of floats in the file (i.e., the number of subsequent lines), as well as the minimum and maximum to-be-sorted float value in the file.

For example, here is the header of a valid input file:

```
4 0.0 9.0
```
BASH

This header indicates that the file contains a total of 4 floats that need to be sorted. The smallest float value in the file will be 0.0, and the largest will be 9.0. You may not need to know the

minimum and maximum values to successfully sort the values, but you will need to inform the user about the range of values being sorted.

Included with the starting point code are a few sample test files you can use to test your code. Every line after the first line in the file will contain a single floating point number. These are the float values that must be sorted. For example, a file containing the header we just saw might look like this (after the first line):

BASH

```
0.0
2.1
9.0
5.3
```

4.2. File I/O

For this assignment we'll use functions from the provided readfile library (in the files `readfile.c` and `readfile.h`). You should not change any code in these files. The readfile.h file contains function prototypes for the readfile library. There are function comments in this file that describe each function and a high-level comment describes how to use the library.

Here are the general rules for how to use these functions:

1. Open a file by calling `open_file()`, passing in the name of the file to open as a string. The return value of `open_file()` tells you whether or not the file was opened successfully. It eturns `0` if the file is successfully opened, and `-1` if the file cannot be opened. You should always check the return value of this function and respond appropriately!
2. Call the `read_int()` `read_string()` `read_float()` functions to read values from the file into your program's variables, where the name of the function you call determines the resulting type of the value that gets filled in. Like `open_file()`, these functions return `0` on success and you should always check their return value. If you've reached the end of the file, they will return `-1`

   These functions take arguments much like `scanf` (https://diveintosystems.org/book/C1-C_intro/input_output.html#_scanf) does. They need to know the memory location of where to put the value read in. For example:

C

```
int x
float f
char s 20];

/* These functions return 0 on success or -1 if read fails or
 * if there is nothing left to read (end-of-file has been reached). */
ret   read_float &f);  // note the & before the variable f
ret   read_int &x       // note the & before the variable x
ret  read_string

/* Note: this is not the sequence of function calls you need
 * to make in your lab solution, but is an example of how to call the
 * three different read functions in this library. In particular, you don't
 * need to use `read_string` for this lab assignment.) */
```

3. Close the file when you're done with it: `close_file()`

If you are curious, the implementation of these functions is in `readfile.c` . You can open this file in your editor to see how it uses the C `FILE *` interface and `fscanf` functions for reading. We will use this interface directly later in the semester. For now, we're hiding it under a layer of abstraction!

5. Lab Requirements

For full credit, your solution should meet the following requirements:

> When run, your program's output should look like the output shown below from a run of a working program. This example doesn't show every possible run or error handling, but should give you some idea of what a correct program run looks like. To make our job of grading easier, please make your output match the example as closely as possible. If you need to print just a new line, use: `printf("\n");`).

BASH

```
$ ./sorter floats.txt
Reading file named floats.txt
The file has 10 floats ranging from 0.00 to 9.00
The unsorted values are: 9.00 8.00 4.00 5.00 7.00 3.00 2 00 1.00 6.00 0.00
The sorted values are: 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00
```

> As in the example above, your program should print the following four items in a human-readable way:
>
> 1. The name of the file
> 2. The contents of the file's header (number of values, max/min values).

3. The values in their original, unsorted order.

4. The values in sorted order, from smallest to largest.

You should store the set of floating point values only once (e.g., keep only one array, don't copy all the values to a second array.) You may assume that there will never be more than 100 values to sort. You may implement any sorting algorithm you'd like, it doesn't need to be fancy. You should not attempt to implement merge sort, as we haven't yet discussed all the C functionality that you need to implement it.

If you don't remember the sorting algorithms we covered in CSC 230, here is the bubble sort algorithm.

```
bubbleSort(array)
  for i in 1 to sizeOfArray - 1
    for j in 1 to sizeOfArray - 1 - i
      if array[j] > array[j+1]
        swap array[j] and array[j+1]
end bubbleSort
```

Your program should exit gracefully if you detect an error that's not your code's fault (for example, if it was given the name of an invalid input file). Be sure to check those return values!

It should be evident that you applied top-down design when constructing your submission. In addition to `main()`, you should have at least two non-trivial functions, each with a specific role. (See the Implementation Recommendations section below for suggestions.)

Your code should be commented, modular, robust, and use meaningful variable and function names. Each function should include a brief description of its behavior.

Write good comments, wrap long lines, use good indenting styles and meaningful variable and function names in your program.

Your program should not use global variables. If a function needs a value from its caller, then that value should be passed in as a parameter. If a function needs to store variables, then those should be declared as local variables inside the function. If you don't know what a global variable is, no problem, you won't make the mistake of using one!

Your code should compile cleanly with no errors or warnings and should not crash unexpectedly during execution.

Code you submit should have "TODO" comments removed. These our notes to you as reminders of some parts to add or where to put parts of your solution in the file. As implement these, remove our "TODO" reminder comments.

5.1. Implementation Guidelines

The following is the suggested way to implement the lab and fulfill the requirements:

Store the floating point values in an array of C `float` types.

Write a function that prints the contents of an array of floating point values. You'll use this twice: first to print the original unsorted array of values, and later to print the sorted array of values.

Write a sort function that takes an array of floating point values (and the size of the array), and sorts the array values such that the original array is in sorted order after the call completes.

You will likely find it useful for your sort function to call two other helper functions that you define: a swap function (to swap the contents of an array, given two indices), and a check function (to determine whether or not your objective has been achieved). Feel free to add your own helper functions too!

6. Tips and Hints

Before even starting to write code, use top-down design to break your program into manageable functionality.

Test your code in small increments. It's much easier to localize a bug when you've only changed a few lines.

Many functions that deal with arrays need to know the size of the array. Unlike Python, C does NOT keep that information around (there is no `len()` function). If a function needs to know the size of the array, you should make it a parameter to the function.

When printing floating point values, use formatting directives to make the output more readable by limiting the number of digits that get printed after the decimal point. For example, using `%.2f` in the format string for `printf()` limits the printed float to two digits after the decimal.

For values that will never change in your program, you can compile in constants using `#defines`. There are some examples with the starting point code.

Use CTRL-C to kill a running program stuck in an infinite loop.

To debug programs with command line options in gdb, include the command line arguments when you start the program with the gdb `u` command:

```
$ gdb ./sorter
(gdb) break main
(gdb) run floats.txt            # run with command line args
```

Post in a discussion and/or come to student hours if you have questions!

7. Submitting your Lab

Please remove any debugging output prior to submitting.

Submit sorter.c to Canvas. Only one partner needs to submit.

At this point, you should submit the required Lab 1 Questionnaire (https://forms.gle/hfCMkRgVwsvcEU259) (each lab partner must do this).

8. Handy References

Class EdSTEM page (https://edstem.org/us/courses/44217/discussion/) for questions and answers about lab assignment

CS31 github org (https://github.swarthmore.edu/CS31-F23)

Git help (https://www.cs.swarthmore.edu/git) and Git for CS31 labs (https://www.cs.swarthmore.edu/~kwebb//cs31/resources/labsetup.html)

Week 1 Weekly Lab Examples (https://www.cs.swarthmore.edu/~kwebb/cs31/f23WeeklyLabs/week1/)

Dive into Systems, Chapter 1 and 2 on C programming (https://diveintosystems.org/antora/diveintosystems/1.0/C_intro/index.html)

Dive into Systems, Chapter 3 on gdb (https://diveintosystems.org/antora/diveintosystems/1.0/C_debug/index.html)

GDB guide (https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.php)

Some Useful Unix Commands (https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_unixCommands.html)

vi (and vim) quick reference (https://www.cs.swarthmore.edu/~newhall/unixhelp/viquickref.pdf)

compiling and running C programs on our system (https://www.cs.swarthmore.edu/~newhall/unixhelp/gccbasics.html)

C code style guide (https://www.cs.swarthmore.edu/~newhall/unixhelp/c_codestyle.html)

make and makefiles (https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html)