Lab 6 – C++ and Basic Sort
CPSC 1021- Fall 2019

CLEMSON
School of COMPUTING

# Due Date: 13 October, 2019 @ Midnight

## Introduction

In today's lab, we will read in a file using C++ and then perform a basic Bubble Sort

## Lab Objectives

- C++ File I/O
- Reading and writing to and from files
- Write a bubble sort routine

## Prior to Beginning This Lab

Make sure you can read in and write out to files in C++.
Read more here: http://www.cplusplus.com/doc/tutorial/files/

## Lab Instructions

Create two files, **functions.cpp** and **driver.cpp.** I am providing you with **functions.h and a sample data.txt**

You will be given an input file that contains the data for a group of students. The name of the input and output file will be determined by command line argument.

You may only use C++ style file I/O to manipulate data. **You may not use printf or scanf in your program**. Additionally, if you use the C++ sort() function, you will receive a zero.

### Your Objectives

- Write a function that reads the data from the input file.
- Write a bubble sort function, sorting the student data by last name in **ascending order**
- Write a bubble sort function, sorting the student data by last name in **descending order**
- Write a function that prints the sorted data.

Example data file:

Yvon Feaster 10 7 1963
Toby Dean 10 16 1962
Mitch Rap 12 15 1959

All data will be in this format.

## Bubble Sort

Bubble sort is a simplistic **sorting** algorithm which allows us to rearrange the elements of our array into sorted order in **$O(n^2)$ time**. This **big-oh notation** tells us the **upper bound time complexity of our algorithm.** All that really means is, given some problem which has a size **n**, the time it takes to solve the problem is no worse than order of **$n^2$.**

For example, if we had 10 numbers to sort, the time complexity of bubble sort tells us that it will require somewhere in the ball-park of 100 operations to completely sort them. This description is simplistic and incomplete, but it does capture the main idea: higher time complexity means higher worst case time to solve a problem. Notice how big $n^2$ becomes as n grows large (n = 1,000 → $n^2$ = 1,000,000). The best comparison based sorting algorithm **quicksort** has time complexity O(n log(n)), which is much fast as n grows large.

For **bubble sort**, the idea is we scan through the array and allow large values to "bubble up" to the end of the list.

See: https://en.wikipedia.org/wiki/Bubble_sort

The basic pseudo-code for this algorithm is as follows:

```
bubble_sort( array, length ) {
  while ( number_swaps != 0 ) {
    number_swaps = 0
    for ( index, index < length - 1, index++ ) {
      if array[index] > array[index+1] → swap; number_swaps++
    }
 } }
```

We keep looping through the array until we do not perform any swaps. At the end of the algorithm, the numbers are sorted from left to right, smallest to largest.

**Write a function that implements this algorithm.**

## Header Files and Header Guard

Each function should have its prototype defined in the functions.h file. At this point, we should be following best practice for maintaining our source code, and using **header guards.**

https://en.wikipedia.org/wiki/Include_guard

This protects us from multiple definition errors when compiling.

You should also comment your code similar to the example given below.

Before each function**, in the functions.h file**, you should have a detailed description of what the overall function does. To borrow from another student's code, here is an example of overall function description.

```
/* Parameters: img - image_t pointer array holding the image data for
 *                   each of the input files
 * Return:       output - image_t struct containing output image data
 * This function averages every pixels rbg values from each of the
 * input images and puts those averages into a single output image
 */
```

You are required to have this type of comment block before each function.
Also, if you include comments in the body of the function (and you should) they should be placed above the line of code not beside the code.
Example:

Good

```
//This is a comment
if(something)
{
    do something;
}
```

Bad

```
if(something) //This is a comment
{
    do something;
}
```

## What to turn in

- **functions.cpp** which provides the implementation for the needed functions.
- **functions.h** which has the prototype
- **driver.cpp**
- **makefile** – the make file should provide a make run and make clean. The input file name should be **data.txt** and output should be **out.txt**

## Compile and Execute

Use g++ to compile your code as follows:

*g++ -Wall functions.cpp driver.cpp -o driver -std=c++11*

Execute the program

*./driver <input> <output>*

## FORMATTING:

1. Your program should be well documented
2. Each file should include a header:
3. Your program should consist of proper and consistent indention
4. No lines of code should be more than 80 characters

```
// Sample Header
/******************
  your name
  username
  Lab 1
  Lab Section:
  Name of TA
******************/
```

5 – 10 points will be deducted for each of the above formatting infractions.

## Submission Instructions

- Test your program on the School of Computing server prior to submitting.
- Use the tar utility to tar.gz all source files. **Do not tar an entire directory! When I untar your archive, I should see all the files you included, not a top-level directory! Failure to correctly tar may result in up to a 25-point penalty!**

  ***EX. tar –czvf yfeaste-lab6.tar.gz <list the files you are tarring>***

- Name your tarred file **<username>-lab<#>.tar.gz** (ex. yfeaste-lab6.tar.gz)
- Use handin (http://handin.cs.clemson.edu) to submit your archive