ECE 150: Fundamentals of Programming (Fall 2016)

Assignment 5

Due on Tuesday, November 8, 11:59pm ET
Total points: 25

This assignment is intended to have students practice creating C++ functions, passing parameters by value and by reference, passing arrays, and passing pointers, as well as taking inputs as command-line arguments. Other previously learned programming concepts such as selection, loops, and arrays are also needed and will be reinforced in this assignment.

Learning Outcomes

After completing this assignment, students should be able to achieve the following:

1. Create C++ programs that take user input as command-line arguments
2. Develop C++ functions that pass parameters by value and by reference, as well as passing arrays and pointers
3. Reinforce previously learned C++ syntax and programming concepts

Task 1 — Find Statistics in Functions (7 points)

Submission: You should submit the result of this exercise to the project `150-5-Statistics`. Your submission file must be named `Statistics.cpp` (case sensitive).

Task Description: This is an exercise to input no more than 20 integer numbers into an array using command-line arguments, and then find the maximum, minimum, and average values of the numbers that are stored in the array. Your `main()` function will have the signature:

```
int main(int argc, char* argv[])
```

where `argc` is the number of arguments (including the program name) and `argv` is an array of null-terminated strings, with `argv[0]` containing the name of the program, while `argv[1]` to `argv[argc-1]` are strings for each command line argument. For example, if your program is executed as follows:

```
Statistics 10 13 2 23 89 71 2 16
```

Then `argc` will be `9`, `argv[0]` will contain `"Statistics"`, `argv[1]` will have the value `"10"` (the string "10", null-terminated, not the `int` value 10), `argv[2]` will have the value `"13"`, *etc*.

You will then design and implement the following functions:

(a) `int convertToIntArray(int argc, char* argv[], int array[]);`

It should take the parameters `argc` and `argv`, and convert the values in `argv[1]` to `argv[argc-1]` from strings into `int`, and store those `int`s into the array. If there are no errors, it should return the number of `int`s converted and stored in the array. If there are more than 20 `int`s, or any of the `argv[]` values are not `int`s, you should return a negative number. The negative number you return should be the negative of the number of the argument where the problem occurred, or -21 if there are too many `int`s.

Important Note #1: In writing your program, **you are not allowed to include the text** `string.h` or `cstring` **anywhere in** `Statistics.cpp` (whether it is part of your C++ instructions or a code comment).

(b) `int findMax (int array[], int numElements);`

This function receives the array from part (a) and the number of elements (also from part (a)), and should return the maximum integer among the integers stored in the array.

(c) `int findMin (int array[], int numElements);`

This function receives the array in part (a) and the number of elements, and should return the minimum integer among the integers stored in the array.

(d) `float average (int array[], int numElements);`

This function receives the array in part (a) and the number of elements, and should return the average value of the numbers stored in the array.

Finally, in your `main()` **function, you should output to** `cout` **the results of each statistical result** if there are no errors when executing the `convertToIntArray()` function (*i.e.*, it returns a number > 0). For example, if you enter the following from the command line:

```
Statistics 10 13 2 23 89 71 2 16
```

The corresponding output should look like:

```
Maximum: 89
Minimum: 2
Average: 28.25
```

Important Note #2: If an error occurs when executing the `convertToIntArray()` function (*i.e.*, it returns a negative result), **you should output to** `cerr` **an error message** that specifies the problem. For instance, if the program was called as:

```
Statistics 10 13 2 2# 89 71 2 16
```

Then the output should be:

```
Error: Argument 4 is not an integer.
```

If the problem is too many integers (more than 20), you should output to `cerr` the message:

```
Error: too many input arguments; maximum 20 permitted.
```

Important Note #3: All functions and your `main(int argc, char* argv[])` **routine should be contained within the file** `Statistics.cpp`.

Task 2 — Running-Average Voltage Calculation and Histogram Analysis (8 points)

Submission: You should submit the result of this exercise to the project `150-5-Sliding`. Your submission file must be named `Sliding.cpp` (case sensitive).

Task Description: Homer Simpson is measuring electrical voltages at various homes in Springfield to make sure that the Nuclear Power Plant is generating enough power. The measurements are a bit erratic because every time something electrical is turned on or off (and also because Mr. Burns will only pay for cheap voltage sensors), the load changes and so the voltage changes. Homer would like to apply a (low-pass) filter that will smooth out the voltage measurement results. There are various ways to do this, but for this first attempt Homer plans to use a sliding window. Your job is to design a help Homer develop a C++ program for this purpose. This program will include three parts, as explained below.

(a) You should write a main function that takes in raw voltage values command-line arguments. In other words, it should have the following signature:

```
int main(int argc, char* argv[]) {
     // Main body code
}
```

`argv[0]` would be the executable file name; `argv[1]` would be the sliding window size, while the remaining arguments (`argv[2]` to `argv[argc-1]`) are the raw voltage inputs. For example, if you wanted to execute this program for a window size of 4 and with 10 voltage measurement values, it would be called as follows:

```
Sliding 4 120 123 163 114 155 126 177 118 96 110
```

Within this `main()` function, the following operations should be performed:

1) **Convert numerical strings within the `argv` array into `int` values**, and store them respectively in a variable that stores the sliding window size and an array that stores all the raw voltage readings. You can reuse the `convertToIntArray()` function that you have developed in Task 1 to achieve this operation (you can assume that the voltage measurement values entered into the program are all integers). Also, in line with Task 1, the maximum number of voltage measurements to be handled is limited to 20 (though do not forget about the extra value specifying the size of the sliding window when dealing with command-line arguments!).
2) **Calculate the sliding-window average values** based on the raw voltage data input. This will be done through a call to the function with signature described in (b) (see below).
3) **Draw a histogram of raw voltage input data**, by calling the function with signature described in (c) (see below).
4) **Draw another histogram for the averaged voltage values**, again by calling the function with signature described in (c).

Important Note: Similar to Task 1, **you are not allowed to include the text `string.h` or `cstring` anywhere in `Sliding.cpp`** (whether it is part of your C++ instructions or a code comment). Also, remember that output is displayed to `cout`, while error messages are sent to `cerr` and should include the word "Error."

(b) You are required to write a sliding-window average function with the following signature:

```
int slideAvg(int inputVoltages[],
             int sampleSize,
             int windowSize,
             float slidingAverage[]);
```

This function would take four parameters as input: i) `inputVoltage`, which is an array of voltage values; ii) `sampleSize`, which is the number of samples there are in the `inputVoltage` array; iii) `windowSize`, which is the size of the sliding window to be used for the averaging process; iv) `slidingAverage`, which is an array that stores the averaged voltage values. As output, this function would return the number of entries present in the `slidingAverage` array (or a negative number if an error occurred).

A sliding window averaging method works like this. Given a series of numbers, a running average can be obtained by averaging a subset of adjacent input numbers. For example, if an input set of 10 numbers is:

120, 123, 163, 114, 155, 126, 177, 118, 96, 110

The running average of a window (or subset) of size 4 will be the averages of:

120, 123, 163, 114 ➔ Average = 130
123, 163, 114, 155 ➔ Average = 138.75
163, 114, 155, 126 ➔ Average = 139.5
114, 155, 126, 177 ➔ Average = 143
155, 126, 177, 118 ➔ Average = 144
126, 177, 118, 96 ➔ Average = 129.25
and 177, 118, 96, 110 ➔ Average = 125.25

Notice that in the above example, the number of running average values that can be generated is 10 – 4 + 1 = 7 (i.e. *number of input values – window size* + 1).

(c) You are required to write a histogram generator function with the following signature:

```
void displayHistogram(int numElements, float array[]);
```

It will create a histogram display of voltage values based on the `array` input that is of `numElements` in size. Basically, this is to please Mr. Burns, who likes Homer's idea (because it saves him from buying more expensive sensors) but prefers to look at pictorial plots rather than numbers in the output. In particular, Mr. Burns would like the program to display a histogram of the voltages, both before and after the filter is applied.

To approach this task, let us not forget that, in North America, household voltages are nominally 120V (±5%), though it is not unreasonable to expect larger fluctuations, and so a histogram that covers the range from 100V to 140V is reasonable. Thus, we can use simple ASCII graphics for our histogram as follows. The range of the histogram will be shown at the top, with the line

```
100V------105V------110V------115V------120V------125V------130V------135V------140V
```

If you look closely at this line, you will notice that there are exactly 10 ASCII characters between the "1" of the "100V" and the "1" of the "105V". It is likewise for every point in that line. Therefore, we will use a "**#**" for each 0.5 V value, starting with a single "#" under the "1" of the 100V to represent 100V. For example, the following histogram output:

```
100V------105V------110V------115V------120V------125V------130V------135V------140V
####################
##########
##########
##############################
############
############
##############
############################
##
```

represents the voltages 110, 105, 104.5, 115.5, 106, 106, 107, 114.5, and 100.5. Here are a few rules that should be followed in generating such a histogram:

1) Given that the accuracy of the display is 0.5 V, you will need to round the numbers in the array to the nearest 0.5 before displaying them.
2) While the numbers *should* be between 100 and 140, to ensure that the function can still output a meaningful display regardless of the data, when an array value to be displayed is less than 100, you should display just a "!"
3) If the array value to be displayed is greater that 140, you should display not only the 81 "#"s that would represent 140V, but a "+" after that.

Task 3 — Sort Names (10 points)

Submission: You should submit the result of this exercise to the project `150-5-Sorting`. Your submission file must be named `Sorting.cpp` (case sensitive).

Important Note #1: This task will be the first instance where **you will develop a source code file with only the required C++ functions**. Within your source code file (`Sorting.cpp`), it should have the statement

```
#include "Sorting.h"
```

which we will provide. `Sorting.cpp` **should not include a** `main()` **function**.

Important Note #2: We will also provide a separate file, `SortingMain.cpp`, which will contain a sample `main()` for how your four functions will be used. It is expected that you will change `SortingMain.cpp` to create various tests of your own to test your sorting functions.

Important Note #3: Do not submit `Sorting.h` nor `SortingMain.cpp`. You should only submit `Sorting.cpp` to marmoset for testing.

Task Description: There is often a need to sort on more than one field. For example, given a list of first and last names, we may want the list to be sorted by last name, and within the same last name, by first name. However, since the list may already have been sorted by a different attribute, we do not want to disturb that sort order when names are otherwise identical. An approach to doing this is to use a *stable* sorting algorithm. A stable sorting algorithm has the property that if two keys are equal on input, the entries maintain their relative ordering. It is easy to ensure that bubble sort is stable. We swap two entries only if the first is strictly larger than the second. For this problem, you are given as input to a sorting function an array of last and first names, together with an array of the associated person's age, as an `int`. You are to sort by last name and within the same last name, by first name. "Sort" means non-decreasing in lexicographic order.

Input: An example of these inputs being provided to your function is as follows:

```
char* firstName[] = {"Andrew", "John", "Jean",
                     "Stephen", "Alice", "Jean"};
char* lastName[] = {"Hawking", "Smith", "Smith",
                    "Hawking", "Cooper", "Smith"};
char age[] = {13, 27, 19, 72, 68, 28};

bubbleSort(6, firstName, lastName, age);
```

Output: You should output a list of first and last names sorted, together with the associated age. For the above example, the output should be:

```
Alice Cooper 68
Andrew Hawking 13
Stephen Hawking 72
Jean Smith 19
Jean Smith 28
John Smith 27
```

You need to write the following functions:

(a) Bubble sort, which should have the following signature:

```
void bubbleSort(int numNames, char* firstName[],
                char* lastName[], int age[]);
```

The function receives an integer representing the number of names (where a name consists of the first name and the last name), the two `char` arrays, and the `int` array. It should use the functions in parts (c) and (d)

(see below) to sort the names (and move the ages as necessary to ensure it is still associated with the same name) as described above according to the bubble sort algorithm.

(b) Output display, which should have the following signature:

```
void displayNames(int numNames,
                  char* firstName[],
                  char* lastName[],
                  int age[]);
```

This function is to receive an integer representing the number of names, the two `char` arrays, and the `int` array, per part (a). The function should print out to the console the first name and last name pairs, followed by the age, one name per line as shown in the sample output above.

(c) String comparison, which should have the following signature:

```
int stringCmp(char str1[], char str2[]);
```

This function is to receive two null-terminated character strings, and compare them. If the strings are identical, it returns 0. If `str1` is lexicographically earlier than `str2` then the function returns +1. Otherwise it returns -1. For example:

```
stringCmp("smith", "smith")
```

returns 0. Likewise

```
stringCmp("smith", "jones")
```

will return -1. For strings of unequal length, that are otherwise equal, the shorter string is considered to be lexicographically earlier. Thus,

```
stringCmp("smith", "smithfield")
```

returns +1. Finally, you should assume the comparison is case sensitive, and so

```
stringCmp("Smith", "jones")
```

will return +1 because Capital-S precedes LowerCase-J, even though J precedes S in the alphabet.

(d) String swapping, which should have the following signature:

```
void stringSwap(char* &str1, char* &str2);
```

This function receives two null-terminated character strings (by reference) and swaps them.

Note that, similar to Tasks 1 and 2, **you are <u>not allowed</u> to include the text** `string.h` or `cstring` **anywhere in** `Sorting.cpp` (whether it is part of your C++ instructions or a code comment).