Lab 5: Fun with Strings

Assigned: *Tuesday, November 3rd, 2020*
Due: *Monday, November 9th, 2020*
Points: *100*

- There is NO MAKEUP for missed assignments.
- We are strict about enforcing the LATE POLICY for all assignments (see syllabus).

Introduction

The assignment for this week will utilize concepts of **string manipulation** and **sorting algorithms**.

We will be looking for (and grading) programming stylizations, such as proper use of comments, tab indentation, good variable names, and overall block and function designs. **We will also be grading for meeting requirements, using "class legal" code, and plagiarism** (consequences are serious and dire if you do not follow these rules). So, it is not enough for your lab to just pass the Gradescope autograder! Please read the instructions herein carefully.

It is HIGHLY RECOMMENDED that you develop the algorithms for each program FIRST and THEN develop the C++ code for it.

I have given you requirements (have-to-dos) and hints (not required, just there to help you) for each exercise.

Step 1: Getting Ready

First, open a terminal window and log into the correct machine. Change into your CS 16 directory, create a **lab05** directory and change into it. Remember that at any time, you can check what directory you are currently in with the command **pwd**.

Step 2: Create and Edit Your C++ Files

This week, you will need to create **two (2) C++ programs called sentence.cpp** and **anagrams.cpp**. They are worth 50 points each and have to be submitted properly for full assignment credit.

IMPORTANT NOTE: As usual, this lab will be also graded for use of comments and style. See below for details. In addition, we will take *major* points OFF if you use C++ instructions/libraries/code that either (a) was not covered in class, or (b) was found to be copied from outside sources (or each other) without proper citation.

sentence.cpp

The program will ask the user for a sentence and will then, re-arrange the letters in the sentence by
(a) sorting them in alphabetical order (per ASCII codes) and
(b) removing all non-alphabetical characters.

The program will then show the frequency of every letter in the string (but just the alphabetical characters). You have to distinguish between upper and lower-case alphabet characters.

For example, see the 3 runs below:

```
$ ./sentence
Enter sentence: Hello I love you BABY1125!@!@$
Sorted and cleaned-up sentence:ABBHIYeellloooouvy
A: 1
B: 2
H: 1
I: 1
Y: 1
e: 2
l: 3
o: 3
u: 1
v: 1
y: 1

$ ./sentence
Enter sentence: How now brown cow?
Sorted and cleaned-up sentence:Hbcnnoooorwwww
H: 1
b: 1
c: 1
n: 2
o: 4
r: 1
w: 4

$ ./sentence
Enter sentence: $1,234.00
Sorted and cleaned-up sentence:
```

You should test your program with multiple examples before you submit it. **Make sure your outputs match the above**. The new line between each run example above is NOT part of the program. The strings printed by the program should include a newline at the end, but no other trailing whitespace (whitespace at the end of the line).

REQUIREMENTS – PLEASE READ:

1. Your program MUST utilize the ***Bubble Sort*** algorithm (not another sorting algorithm) that we reviewed in lecture (and also found in Ch. 7 of your textbook). You will need to adapt the algorithm to a string, rather than to an array of integers.

2. Be sure to utilize ONLY techniques we've covered in lecture. Do not use "special" arrays or pointers, do not use vectors, do not use built-in sorting functions, etc… You will get zero points if you do.

3. Start your program using the **sentence_skeleton.cpp** program that I've provided you with. Of course, you will have to rename it **sentence.cpp**. The skeleton program shows 2 called functions, which, of course, you must use. You can create more functions in your program if you need to (you likely will).

HINTS:

1. Realize that, in C++, assigning char types their ASCII code (which are integers) is legal. For example `char letter = 65;` makes **`letter`** be '**A**'. See https://simple.wikipedia.org/wiki/ASCII for a list of ASCII character codes.

anagrams.cpp

This is a program that determines if 2 strings are anagrams. An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, for example, the letters in the word "listen" can be re-arranged to spell "silent".

The function should **not** be case sensitive and should **disregard** any numerals, punctuation or spaces (i.e. any non-alphabets). Two strings are anagrams if the letters can be rearranged to form each other. For example, "Eleven plus two" is an anagram of "Twelve plus one". Each string contains one "v", three "e's", two "l's", etc. You may use C++ strings and arrays to solve this problem.

Write a program that inputs two strings and calls your function to determine whether or not the strings are anagrams and prints the result.

A session should look like one of the following examples (including whitespace and formatting). The user input is shown bolded for your convenience here.

```
$ ./anagrams
Enter first string:
Eleven plus two
Enter second string:
Twelve plus three
The strings are not anagrams.

$ ./anagrams
Enter first string:
Rats and Mice
Enter second string:
in cat's dream
The strings are anagrams.
```

You should test your program with multiple examples before you submit it. **Make sure your outputs match the above**. The new line between each run example above is NOT part of the program. The strings printed by the program should include a newline at the end, but no other trailing whitespace (whitespace at the end of the line).

REQUIREMENTS – PLEASE READ:

1. Be sure to utilize ONLY techniques we've covered in lecture. Do not use "special" arrays or pointers, do not use vectors, do not use built-in sorting functions, etc… You will get zero points if you do.

2. Start your program using the **anagrams_skeleton.cpp** program that I've provided you with. Of course, you will have to rename it **anagrams.cpp**. The skeleton program shows called functions, which, of course, you must use. You can create more functions in your program if you need to.

HINTS:

1. You can create "counting arrays", that is integer arrays that keep track of how many of what letters are found in each input string. For example, array1 has 26 elements, each representing a letter in one of the strings, etc… This is not the only approach that you can use, of course.

Step 3: Create a makefile and Compile the Codes with the make Command

In order to learn another way to manage our source codes and their compilations, we will first create a **makefile** and put in the usual g++ commands in it. Afterwards, whenever we want to compile our programs, the Linux command is a lot shorter – so this is a convenience.

Using your text editor, create a new file called **makefile** and enter the following into it:

```
all: sentence anagrams

sentence: sentence.cpp
	g++ -o sentence sentence.cpp -Wall -std=c++11

anagrams: anagrams.cpp
	g++ -o anagrams anagrams.cpp -Wall -std=c++11
```

Then from the Linux prompt, you can issue one command that will compile this code, like so:

```
$ make
```

If the compilation is successful, you will not see any output from the compiler. You can then run your program like this:

```
$ ./sentence
```

or

```
$ ./anagrams
```

If you encounter an error, use the compiler hints and examine the line in question. If the compiler message is not sufficient to identify the error, you can search online to see when the error occurs in general.

Remember to re-compile the relevant files after you make any changes to the C++ code.

Step 4: Submit using GRADESCOPE

BEFORE YOU SUBMIT YOUR FINAL VERSION FILES: Make sure that you:

a) **STYLIZE** your program appropriately to additionally make it easier on others reading your code. Apply ALL the style pointers I have posted on Piazza. This includes the use of comments.

This will be graded for an *extra 20 points* beyond the automatic grade you get from Gradescope.

b) **MEET THE REQUIREMENTS OF THIS LAB**. This means, if I said certain features or program aspects were *required* to be in your final code, then they must be there.

If these requirements are not met, you will lose MAJOR points on the exercise. This simulates a real-world situation (e.g. your customer wants certain features, so you must provide it – while I am not your customer, per se, I *am* giving you points for your efforts!)

Log into your account on **https://www.gradescope.com/** and navigate to our course site. Select this assignment. Then click on the "Submit" button on the bottom right corner to make a submission. You will be given the option of uploading files from your local machine or submitting the code that is in a github repo. Choose the first option and follow the steps to **upload BOTH C++ files** to Gradescope.

You may submit this lab multiple times. You should submit only after you test it on your computer (or CSIL) and are satisfied that the programs run correctly. The score of the last submission uploaded before the deadline will be used as your assignment grade.

Step 5: Done!

Once your submission receives a score of 100/100 on the autograder, you are done with this assignment. REMEMBER that there are 20 additional points that will be scored according to your proper use of comments and styling. The total will then be normalized to 100 points again (i.e. 120 score will be 100%) to grade this lab.

WE WILL BE CHECKING FOR PLAGIARISM – DO NOT COPY FROM OTHER STUDENTS OR FROM SOURCES ONLINE! USE PROPER CITATION OF CODE THAT YOU DID NOT WRITE! THE CONSEQUENCES WILL - AT MINIMUM - BE A ZERO ON THIS LAB AND POSSIBLY A ZERO IN THIS COURSE AND YOU WILL BE REPORTED TO THE UNIVERSITY.