Lab 6: File I/O

Assigned: *Tuesday, November 10th, 2020*
Due: *Monday, November 16th, 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 file I/O data streams and string manipulation. Again, 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. So, it is not enough for your lab to 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.

Step 1: Getting Ready

First, open a terminal window and log into the correct machine. Change into your CS 16 directory, create a **lab06** directory and change into it. There are no skeleton files provided for you.

Step 2: Create and Edit Your C++ Files

This week, you will need to create **2 C++ programs** called **stats.cpp** and **rhymes.cpp**. Each of the C++ programs corresponds to one of the problems listed below, which make up this lab. Each is worth 50 points and should be solved in its own file and both must be submitted for full assignment credit.

IMPORTANT NOTE: 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.

stats.cpp

This program takes its inputs from a file that contains an indeterminate number of floating-point numbers. The program reads them in as type double. The program outputs to the screen the *mean* (average), the *median*, and the *standard deviation* of the numbers in the file. The file contains nothing but numbers of type double separated by blanks and/or line breaks. For this exercise, the average is the sum of all the numbers divided by the count of these numbers. The median is the number that is in the middle of the list, where ½ of the numbers in the list are smaller than the median and the other ½ of the numbers is larger than the median. If there are an even number of numbers, then the median is the average of the 2 middle numbers. The standard deviation of a list of numbers **x1**, **x2**, **x3**, and so forth, is defined as:

$$\sqrt{((\boldsymbol{x1}-\boldsymbol{a})^2+(\boldsymbol{x2}-\boldsymbol{a})^2+(\boldsymbol{x3}-\boldsymbol{a})^2+\cdots)\,/\,(\boldsymbol{n}-\boldsymbol{1})}$$

Where the number **a** is the average of the numbers **x1**, **x2**, **x3**, and so forth, and the number **n** is the count of how many numbers there are.

REQUIREMENTS, ASSUMPTIONS:

1. Your program should take file name as an input from the user – it does **not** need to check if the input file name is valid.
2. The answers should be given with exactly 3 decimal points.
3. You can safely assume that the number files will never have more than 1000 numbers in them.
4. Your program should define **at least 3 functions** that only do the calculations for the mean, median, and standard deviation (they must **not** print to std.out) and return the values back to the function caller.
    a. You can have additional functions, if you like.
    b. If your program does NOT have these 3 functions, you will not get credit for this part of the assignment, even if your program passes the Gradescope autograder.
    c. You can pass the output of the mean function into the standard deviation function, if you want (would be helpful).
5. 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. You HAVE to calculate mean, median, and standard deviation using the "classical" methods talked about here – no short cuts.

Let's say the input file, **nums1.txt**, contains this (note the separation by whitespaces):

```
6 9
7
8
```

Then, a session should look ***exactly*** like the following example (including whitespace and formatting - note that there is no whitespace at the end of each of these lines and each printed line has a newline at the end). Note the user input is bolded for your convenience.

```
Enter filename: nums1.txt
Mean = 7.500
Median = 7.500
Stddev = 1.291
```

Let's say that another input file, **nums2.txt**, contains this (note the separation by whitespaces):

```
1.2 4.2 0 10 5
```

Then, a session should look ***exactly*** like the following:

```
Enter filename: nums2.txt
Mean = 4.080
Median = 4.200
Stddev = 3.900
```

One more example: consider input file, **nums3.txt** which contains:

```
55.8 86.9 2.4 16.8 5.9 10.2 11.2 25 96.1 4 12.7 81.7 60.4 72.1 52.1 39.6 24.2 2.6 52.3 49.9 14.4
90.9 48.1 88.6 44.3 51.1 55.1 77 33.4 8.4 66.2 89.2 95.3 68.6 41.2 36.5 14 52.4 96.7 10.2 91.7 44.6
27.1 52.1 16.7 79.2 91.7 76.1 81.8 79.3
```

Then, a session should look like this:

```
Enter filename: nums3.txt
Mean = 49.676
Median = 51.600
Stddev = 30.633
```

You will only be submitting the C++ file, **stats.cpp** (there is no skeleton or starter file). We will test it with several different number files.

rhymes.cpp

Write a program that finds *rhyming words in a poem*!

Specifically, the program reads an input file, line by line, and extracts the last word in each line. It then compares this last word with the last word from the line after it. If it finds that the 2 words "rhyme" (which is simply defined as the 2 words that are being compared *share the same 2 last letters*), then it prints out these two words to standard output (i.e. the display).

It also displays the number of lines in the poem along with a metric called "the rhyme-line density" which is the number of rhymes found divided by the number of lines in the poem.

But FIRST, the words must be cleaned up of any non-alphabetical characters (usually this means punctuation, since many poems have their last lines often end with a comma, a semicolon, a question-mark, etc…) So, if the last word on a line has punctuation marks on it, e.g. "hello?!", it becomes "hello". Or if the last word is "don't", it becomes "dont".

Finally, the program has to state how many rhyming pairs it found, or if it did not find any at all.

REQUIREMENTS:

1. Your program should take file name as an input from the user – it **does** need to check if the input file name is valid.
    a. If it does not exist, the program must output (via **cerr**) an error message: "**Input file opening failed.**" and then exit with code 1 (careful: not return 1).
2. Your program should define **at least 3 functions** that only do last-word extraction, word-end comparisons, and word clean-up.
    a. You can have additional functions, if you like.
    b. If your program does NOT have these 3 functions, you will not get credit for this part of the assignment, even if your program passes the Gradescope autograder.
3. When displaying the rhyme-line metric, only show 2 places after the decimal point.
    a. If there are no rhymes found, you do not need to show the metric (see examples below).
4. 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.

As an example, assume there is a text file called "**MyPoem.txt**", which contains this:

```
"I cannot go to school today,"
Said little Peggy Ann McKay.
"I cannot go to school and play!
I have the measles and the mumps,
A gash, a rash and purple bumps.
My mouth is wet, my throat is dry,
I'm going blind in my right eye.
My tonsils are as big as rocks,
I've counted sixteen chicken pox"
```

The program would run as follows. Note the user input is bolded for your convenience.

```
Enter file name: MyPoem.txt
today and McKay
McKay and play
mumps and bumps
There are 3 pairs of rhyming words.
There are 9 lines in this poem, so the rhyme-line density is: 0.33
```

Note that, in this poem, “dry” and “eye” were not shown, nor were “rocks” and “pox”. This is because they do not meet *the program’s criteria of a “rhyme”*. Also note the last line of the output says how many pairs of rhyming words there are. **Please make the word “pair” singular if only 1 pair of rhyming words is found and use the plural “pairs” if you found more than 1 rhyming pairs.**

So, for example, with a file called “**Simple.txt**”:

```
I do not like green eggs and ham,
I do not like them Sam I am.
```

The program would run as follows:

```
Enter file name: Simple.txt
ham and am
There is 1 pair of rhyming words.
There are 2 lines in this poem, so the rhyme-line density is: 0.50
```

Finally, consider this example file called “**No.txt**”:

```
No means no, it always means no
If I say it once it means
A thousand times no!
```

The program would run as follows:

```
Enter file name: No.txt
No rhymes found.
There are 3 lines in this poem.
```

Note that no rhymes were found because the program only looks at adjacent lines. Also, note that the case of no rhymes found will not display the rhyme-line metric.

You can safely assume that:

a. Text files may be empty (have no line), and may contain blank lines, but lines that are not blank will have at least 2 words per line.
b. The last words on a line contain *at least* 2 characters.
c. The rhyme comparison is case-sensitive.

You will only be submitting the C++ file, **rhymes.cpp** (there is no skeleton or starter file). We will test it with several different rhyme files.

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: stats rhymes

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

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

Then from the Linux prompt, you can do one of two things: either issue separate compile commands for each project, like so:

```
$ make stats
```

Or, you can issue one command that will compile all the projects mentioned in the **makefile**, like so:

```
$ make
```

If the compilation is successful, you will not see any output from the compiler. You can then run your programs, for example:

```
$ ./stats
```

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.