Assignment 04: C++ Streaming File Input / Output and I/O Manipulators

COSC 1437: Programming Fundamentals II

Objectives

- Continue practicing using looping constructs for counter controlled and EOF-controlled loops
- Learn about opening and closing input file streams
- Learn how to read in data from files
- Practice performing operations on file data, for example calculating averages
- Learn how to open and close output file streams
- Practice using I/O manipulators to format output into files

Description

In this lab/assignment we will be learning how to use the C++ streaming library to perform more advanced Input/Output in our programs. You have seen many examples of simple I/O using the `cin` and `cout` streams if you have been using the class materials so far in your studies.

The C++ streaming I/O library is an object-oriented way of doing Input / Output. It uses the concept of a stream that can be connected to a source of data. For example, by default the `cin` stream is connected to the keyboard to stream input from it, and the `cout` stream is connected to the terminal device that the program is running from, where any output sent to this stream will be displayed. As you will see in this assignment, we can open up new stream instances and attach them to files (and other data sources). We will use this both to read in and process from a file using an `ifstream` (input file stream) and to write formatted data back out to a file using an `ofstream` (output file stream).

The first two tasks start with streaming file input, and also give some additional practice using counter-controlled and EOF-controlled loops. The third task is a bit more complicated as we will open up a file for reading and another file for writing, and we will process payroll data in the input file and save the formatted results to the output file.

Overview and Setup

For all assignments, it will be assumed that you have a working VSCode IDE with remote Dev Containers extension installed, and that you have accepted and cloned the assignment to a Dev Container in your VSCode IDE. We will start each assignment with a description of the general setup of the assignment, and an overview of the assignment tasks.

For this assignment you have been given the following files (among some others):

File Name	Description
`src/assg04-tests.cpp`	Unit tests for the tasks that you need to successfully pass
`src/assg04-library.cpp`	File that contains the code implementations, all your work will be done here
`include/assg04-library.hpp`	Header include file of function prototypes
`data/x.txt`	Files used for input in the tasks are in the `data` subdirectory
`output`	Output files will be stored in the `output` subdirectory

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub by accepting the assignment using the provided assignment invitation link for 'Assignment 03' for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment. Make sure that you open the cloned folder and restart inside of the correct Dev Container.
3. Confirm that the project builds and runs, though no tests may be defined or run initially for some assignments. If the project does not build on the first checkout, please inform the instructor.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment.

Assignment Tasks

Lab Task 1 / In-Lab Work : Compute Average of Values in a File Using a Counter

For the first task we will be opening a file using an `ifstream`. There are several files in the `data` subdirectory named something like `task01-test01.txt`. The content of this first test file looks like this:

```
4
3.0
6.0
4.0
5.0
```

In this file the first line of data will always hold a counter of the number of values in the file that need to be read in and processed. There are 4 values in this file that we want to read in, and compute the average of.

The remaining `N=4` lines hold the data, that in this case can be read into a stream and interpreted as `double` formatted values.

Our task is to open up a file, then first read in the `numValues` as an `int` data type from the file. Then you need to create a `for` loop that uses this `numValues` for a counter-controlled loop to read in the remaining `double` values. While reading in the values, you will keep a running sum of the values. And at the end this function computes the average of the values in the file by dividing the sum of the values by the `numValues` that were read in.

As usual start by `#define` the `task1` tests in the `assg04-tests.cpp` file, and also adding the Task 1 issue to your GitHub classroom. We will stop reminding you to do these to start a task, from now on you should begin by enabling your tests and creating your next issue for each task.

In the `assg04-library.cpp` the first function named `averageNValuesInFile()` is where you need to put your code for this first lab task. The name of the file to open is passed in as a `string` data type. If you look at the first test for this task, it calls this function like this:

```
  CHECK_THAT(averageNValuesInFile("data/task01-test01.txt"), Catch::WithinAbs(4.5, 0.000000000001));
```

So notice that we hard code a string literal with the name of the file to open and process in the `averageNValuesInFile()` function. Also notice we use a relative path name here. The test executable runs in the top level directory of your project, so all of the input data files are in the `data` subdirectory relative to where these tests are run.

You need to start all of the tasks in this assignment by opening the given file in the function. Here is an example of how you can do this, to get you going:

```
  // create an input file stream instance, and open up the file passed in as the
  // inDataFilename parameter to this function
  ifstream inData;
  inData.open(inDataFilename);
```

Once you have the file open and attached to the `inData` stream, you can use `inData` just like `cin`, that you should have seen many examples of so far in your studies.

To complete the work for this task, perform the following steps:

1. Open up the file for reading using an `ifstream` object as just shown.
2. The first value of this file is the number of values in the file to be processed. Create a variable of type `int` named `numValues`, and read in this first integer value from the file.
3. We need some local variables of type `double` to keep track of the sum and to read the next value into. Create two `double` variables named `sum` and `nextValue`. Make sure you initialize `sum` to be `0.0` before we started adding in values to this sum.
4. Create a `for` loop. The loop needs to execute exactly `numValues` times, so that it will read in that number of values from the file and process them.
5. In the body of the loop you need to
   - First read in the `nextValue` from the file.
   - Then add that value just read into the running `sum`
6. After the loop is done we no longer need the file. It is always best practice to close any file you open as soon as you are done with it. So close the `inData` file stream now.
7. The function will be hard coded to return `0.0` when you start this task. Instead calculate the average of the values from the file (e.g. the `sum` divided by the `numValues` will be the average of the values), and return the average result you calculate using a `return` statement.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 1 pass successfully. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

Task 2: Compute Average of All Values in File Using EOF Controlled Loop

For Task 2 we will again be computing the average of all of the values in some file. However, whenever possible, we almost always just want to process all of the values in the file. We don't know how many values there are, but we need to read in all of them and compute the average of all values we find. The name of the function you will be modifying is `averageAllValuesInFile()` in the `assg04-library.cpp` file.

This means we want to use a type of sentinel-controlled loop structure here, in particular since we want to keep reading and processing until we reach the end-of-file (EOF), we will use what our Malik textbook calls and EOF-controlled loop.

The format of the data files for task02 then is similar to before, but there is no indication of the number of values to process as the first value in the file (you can for example open `data/task02-test01.txt` to see what it looks like).

You are required to use a `while` loop and test when you reach the end-of-file to read in all `double` values from the file and sum them up and average them. Do the following steps for this task:

1. Open up the file for reading using an `ifstream` as you did for the first task.
2. You will need a `numValues`, `sum` and `nextValue` local variable again. But this time you need to count the `numValues` you read in, so this variable should still be an `int`, but you should initialize it to 0 before performing your loop.
3. Use a `while` loop that keeps processing the file until you detect you have reached the `eof()`.
4. Inside of the loop you need to perform the same tasks as before, read in the next value, then add it to the running sum. But you also have to update the `numValues` by incrementing it each time you successfully read in a value.
5. As with the previous task, once the loop is done you should close your file since you no longer need to process it.
6. And you should be able to calculate and return the average value in the same way you did before, using the `sum` and the `numValues` you processed.

When finished, save compile and rerun the tests. Once you are satisfied, make and push your commit of Task 2 to your GitHub classroom repository, and check that it compiles and passes tests in the autograder.

Task 3: File Output and I/O Stream Formatting using Manipulators

For the final Task 3 of this assignment, you will be both opening a file to read data from it, but also opening a file using an `ofstream` to format and write back out the results of your processing. You work will be in the `processPayrollRaises()` function in the `assg04-library.cpp` file for this task.

Look at the file named `output/task03-test01.txt`. The first line of input in this file looks like this:

`Doctor Who 65789.87 0.081`

The actual format of this file is that, each line contains data in the following format:

`firstName lastName currentSalary raiseRatio`

We will use `string` types to read in the first and last names from this file, and `double` types to read in the persons current salary and a percentage raise ratio we are to apply to generate a new salary.

The output to be generated looks like this (the first line of the `data/task03-out01.res` file):

`Who, Doctor###############$.71118.85`

You are to output the name as "lastName, firstName" in a field of width 25, left justified, and filled with the `#` character for any unused space in the field. Then a `$` is output. Then the calculated new salary is output in a field of width 9, right justified (so the decimals align) showing only 2 decimal places of precision, and using `.` to fill any any unused space in the field.

To reformat the name you will create a new `string`. So for example if you have read the `firstName` and `lastName` into `string` instances from the file, you can create a new string of the full name like this:

```
string fullName;
fullName = lastName + ", " + firstName;
```

We will actually look at the `string` data type in more detail soon in the class. Here this shows that strings support concatenation using the `+` operator, so we end up with "Who, Doctor" in the `fullName` if `Doctor` was the first name and `Who` was the last name.

You also need to calculate the new salary raise. You should do that simply by applying the `raiseRatio` to the `currentSalary` like this:

```
double newSalary;
newSalary = currentSalary + (raiseRatio * currentSalary);
```

This calculates a percentage raise from the `currentSalary` and then adds that amount onto the current salary to get the new salary.

So with the above in mind, perform the following tasks in the `processPayrollRaises()` function for Task 3:

1. Open up the input file using an `ifstream` as before.
2. Also open up the output file, given as the second parameter to this function, using an `ofstream`.
3. You will need the following local variables to read in and process data from the input file
   - `firstName`, which should be a `string` type
   - `lastName` which is also a `string`
   - `currentSalary` which should be of type `double`
   - and `raiseRatio` which is also a `double`
   - In addition you need a `numEmployees` variable of type `int` that is initialized to 0, in which you will keep track of the number of employee records you process (like you did for `numValues` in the previous task).
4. You also need some variables for processing the employee payroll records
   - As described above, create another `string` named `fullName` where we will put in the full name of the employee, last name first followed by first name.
   - Also create a `double` called `newSalary` to hold the calculation of the new salary amount.
5. Now we are ready for the loop. Use a `while` loop. Have the loop keep running as long as the `inData` stream is NOT at the end of file yet.
   - You should first read in the 4 pieces of data from the input file from the current line, the `firstName`, `lastName`, `currentSalary` and `raiseRatio`.
   - Update the `numEmployees` you have read in by 1.
   - Calculate the `newSalary` using the read in `currentSalary` and `raiseRatio`.
   - Create the `fullName` in the "lastName, firstName" format from the names you have read in.
   - Output the new record to the output file stream that you have opened.
     - The full name should be left justified, in a field of width 25, filling any unused space with the `#` character.
     - This is followed by a `$` character for the new salary

   - Then the new salary is in a field of width 9, right justified and filled with '.' for unused space. It should only show 2 decimal digits of precision since we are displaying a salary in dollars and cents. It should display using `fixed` format not in scientific notation.
   - You need to also output a newline (`endl`) in the stream at end of each employee record output, for the start of the next employee.
6. After the loop ends, make sure you close both the input and the output file stream. This is more important than before here. Closing the output file stream ensures that all data will get flushed and written out to the file.
7. This function should return the number of employees as an `int` result that were processed. So you need to change the `return` statement to return the number of employee records you read and processed.

The tests for this task check that the number of employee records process is correct. But it also does a file `diff` if your ouput file in the `output` subdirectory, and the expected result file in the `data` directory. If your output file does not exactly match, character for character, this expected result, then the test will fail in task 3. If that is happenning, you might want to open your output and the result file and compare them side-by-side to see what differs.

When finished, save compile and rerun the tests. Once you are satisfied, make and push your commit of Task 3 to your GitHub classroom repository, and check that it compiles and passes tests in the autograder.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is OK. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this assignment, up to 50 points will be given for having completed at least the initial Task 1 / In Lab task. Thereafter 25 additional points are given by the autograder for completing tasks 2 and 3 successfully.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull request comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
   - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
   - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.

5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Git Tutorials
- Git User Manual
- Git Commit Messages Guidelines
- Test-driven Development and Unit Testing Concepts
- Catch2 Unit Test Tutorial
- Getting Started with Visual Studio Code
- Visual Studio Code Documentation and User Guide
- Make Build System Tutorial
- Markdown Basic Syntax