Due Date: Mar 23, 2022 @ 11:59:59

File I/O Introduction – Summarize Grades

This assignment will focus on File I/O operations. You will write a program that reads a file with student information and their homework scores. You will manipulate that data to compute the total score and percentage for each student and then write a summary file with the computed scores for each student. Also, you will clean up some of the formatting issues from the input file.

Your will submit your work to Gradescope and it will be both autograded and human graded. The autograder will compile your code with g++-7.5.0.

Be sure to watch Piazza for any clarifications or updates to the assignment.

Don't forget the Assignment 3 Canvas Quiz!
Your assignment submission is not complete without the quiz.

Summarize Grades Overview

Your submission must consist of 3 files: `SummarizeGrades.h`, `SummarizeGrades.cpp`, and `demo.cpp`. The first being the header file, the second being the implementation of the required functions, and the third being the `main()` function implementation for testing purposes.

For `SummarizeGrades.h`:

- Be sure to have include guards.
- Don't import anything that is unnecessary.
- Include file header comments that have your name, student id, a brief description of the library, and be sure to cite any resource (site or person) that you used in implementing the library.
- Each function should have appropriate function header comments.
  - You can use javadoc style (see doxygen) or another style, but be consistent.
  - Be complete: good description of the function, parameters, and return values.
- For this assignment, no classes and no inline methods.
- In `.h`, you need to define a `struct` called `Name`. It has the following members:
  - `string firstName`
  - `string lastName`

For `SummarizeGrades.cpp`:

- All the function bodies for those defined in `SummarizeGrades.h` should be here.
- Don't clutter your code with useless inline comments (it is a code smell [Inline Comments in the Code is a Smell, but Document the Why], unless it is the why).
- Follow normal programming practices for spacing, naming, etc: Be reasonable and consistent.
- Be sure to avoid redundant code.

Functions to implement:

`void readGradeFile(const string inputFile, int *numAssignments, map<int, Name> &studentNames, map<int, vector<int>> &studentScores)` This function takes the path of the input file and references to some data structures that will hold the student names and scores for other functions to process. It also contains pointers to variables that will keep track of the number of students and assignments.

A sample of an unformatted gradebook file is provided for reference in Canvas. The unformatted gradebook file has the following structure:

- The first line has the keyword `number_of_assignments` followed by an integer indicating the total number of assignments for the course
- The second line contains three keywords - `student_number`, `first_name`, `last_name` and then integer values to indicate the maximum possible number of points for each assignment. Note that all assignments will be out of 10.
- The rest of the lines will have data for one student. First will be a 5-digit student number followed by their first and last name. Then, the scores for each of the assignments are listed. You may assume that there are no duplicate records and any file we use to test your work will follow this format exactly; however, the number of students and the number of assignments will vary.

Your code should do the following:

- Open the file for reading and begin processing the file contents:
  1. Read the number of assignments and store that value in the `numAssignments` variable.
  2. Process each of the students:
     - Store the student name in the `studentNames` map with the student ID as the key, and the value as a `Name` structure containing the first name and last name.
     - Store the scores of each student in the `studentScores` map with the student ID as the key, and the value is a vector of ints to hold the scores.
- Close the file when finished.

`void formatCaseOfNames(map<int, Name> &names)` This function takes the student names map and changes the first and last name strings in the struct (values in map) so that the first letter of each string is a capital letter and all other letters are lower case.

`void computeTotalAndPercent(const map<int, vector<int>> &scores, map<int, int> &total, map<int, double> &percent, int numAssignments)` This function takes the `scores` map and computes the total score (the sum of all points earned) as well as the final percentage as a float and stores these computed results in total and percent maps respectively where key is the key from the `scores` map.

For calculating the percentages, you can assume that all assignments are worth 10 points each. Percentages should be computed as doubles. The minimum value should be capped at 0, and the maximum value should be capped at 100.

`void writeFormattedGrades(const string outputFile, const map<int, Name> &names, const map<int, int> &total, const map<int, double> &percent)` This function will output the formatted grades to the file designated in `outputFile`, overwriting any contents if the file exists as follows.

- Each line of the file will begin with the student's last name followed by a comma, a space and then their first name.
- Following the name, will be the total score. Scores should be aligned so that the ones digit of every total falls in the 22nd column. You may assume that the name length (last name + first name) is always $\leq$ 15 characters and the total score will take at most 3 characters.
- Following the total score, will be the percentage score. Percentages should be values between 0 and 100 written to one decimal place of precision. This means that even if the percentage is 100, it will be reported as 100.0. The decimal point goes in the 28th column.

Please see the sample output file `formatted_grades.txt` for an example. We recommend using IO manipulators such as setw and setprecision. Your output file produced when processing the sample input file `unformatted_grades.txt` should look exactly like the sample file `formatted_grades.txt`. You can use diff utility to compare two files.

Remember to close the file when you are finished.

demo.cpp

You must create a program (place it in `demo.cpp`) that demos all the functions in the library. There is no fixed output required, but your `demo.cpp` requires:

- A `main` function.
- At least 1 call to each of the functions from `SummarizeGrades.h`.
- You may use the sample input file in your `demo.cpp`.

Makefile

Your submission must include a `Makefile`. The minimal requirements are:

- Compile using the demo executable with the `make` command.
- Use appropriate variables for the compiler and compiler options.
- It must contain a clean rule.
- The demo executable must be named something meaningful (not a.out).

README.md

Your submission must include a `README.md` file (see https://guides.github.com/features/mastering-markdown/). The file should give a brief description of the program, the organization of the code, how to compile the code, and how to run the program.

Submission

You will upload all the required files to Gradescope. There are no limits on the number of submissions. See the syllabus for the details about late submissions.

Grading

For this assignment, half of the marks will come from the autograder. For this assignment, none of the test details have been hidden, but there may be hidden tests in future assignments.

The other half of the marks will come from human-grading of the submission. Your files will be checked for style and to ensure that they meet the requirements described above. In addition, all submitted files should contain a header (comments or otherwise) that, at a minimum, give a brief description of the file, your name, and wisc id.

Rubric

- Autograder Tests - 40
- README File - 5
  - Name / student id
  - Typeset/organized with markdown
  - Program description
  - Compilation instructions
  - Run instructions
  - Code organization description
- Makefile - 5
  - File header comments including name/id
  - Demo build rule with appropriate named executable
  - SummarizeGrades.o build rule
  - Clean build rule
  - Correct file name
- demo.cpp - 10
  - File header comments including name/id
  - Tests all functions
- SummarizeGrades.h - 10
  - File header comments including name/id
  - Include guards
  - All includes are required for header file
  - Function header comments with function description, return value and parameters
  - Comment detailing other elements like structs
  - Function prototypes include default values as per the specification
  - Only inline function bodies are defined in header
- SummarizeGrades.cpp - 10
  - File header comments including name/id

- No unused/unneeded includes
- Consistent style (braces, spacing, etc)
- Avoid inline comments
- Avoid duplicate code, use helper functions if needed
- Avoid c-style casting
- Avoid gotos
- Well-designed functions
- Using C++ streams for reading and writing files

Tackling Assignment 3

1. Create a directory for your assignment 3 project, and create the required files: `SummarizeGrades.cpp`, `SummarizeGrades.h`, `README.md`, `Makefile`, and `demo.cpp`. Download the sample input and output files from Canvas.
2. Get things compiling: Put the function prototypes in `SummarizeGrades.h`, the function stub details in `SummarizeGrades.cpp`, and set up your makefile. (Pro tip: When adding you function prototypes, also do the function header comment).
3. Implement and test the functions `formatCaseOfNames` and `computeTotalAndPercent` that don't have any file I/O. Put some tests in your `demo.cpp`. When that is working, check if they pass the autograder tests.
4. Reading a file: Implement and test the `readGradeFile` function. Add some tests to `demo.cpp`, and also check if it passes the autograder tests.
5. Writing a file: Implement and test the `writeFormattedGrades` function. Add some tests to `demo.cpp`, and also check if it passes the autograder tests. Be sure to check your output against the sample output file.
6. Don't forget to do the `README.md` file.
7. Final submission: make sure you have file comments, and name on all files. Double check the rubric, and do your final submission to Gradescope.

HAPPY CODING!