CE221 ASSIGNMENT 2 2019/20

Set by: Mike Sanderson

Credit: 20% of total module mark

Deadline: 11.59.59, Wednesday 15 January

Submission of this assignment will be via the online submission system (FASER).

It is expected that marks and feedback will be returned by the middle of week 20.

You should refer to sections 5 and 7 of the Undergraduate Students' Handbook for details of the University policy regarding late submission and plagiarism; the work handed in must be entirely your own. Your programs will be checked by an intelligent plagiarism detection system that looks for similarities between the submitted programs.

Introduction

This assignment involves two independent programming exercises – the first is worth 50% and the second 40%, with the remaining marks being awarded for programming style and comments. The two programs should be developed within separate folders inside an assignment 2 folder. Material for the exercise will be found on Moodle in a zip file.

Exercise 1

Within the folder **`ex1`** in the zip file you will find files called **`Person.h`** , **`Person.cpp`** and **`Student.h`**. These provide a class to hold details of a person, and declarations for a derived class to hold details of a student. Such classes would normally be expected to have more data members but only those that are relevant to this assignment have been declared.

The `Person` class is complete but you may, if you wish, add extra member functions to the class, e.g. an `operator string` method; you should not change any of the existing material.

The `Student` class declaration simply provides declarations for member functions that must be implemented; you must provide complete definitions for these functions in a file called **`Student.cpp`**. The behaviour of the functions should be as specified in the comments supplied in the header file. Again you may, if you wish, add extra member functions to the class (e.g. a method to calculate the average mark); if you do so the function bodies should be written in the **`.cpp`** file. You should not add any extra data members.

In a separate file, write a `main` function that will attempt to open and read from a text file, whose name should be supplied by the user. Each line of the file will contain the registration number and name of a student; these should be passed as arguments to the `Student` constructor to create a new student object,which should be added to a collection of students. This collection may be a `vector`, a `list` or a `set` (from the standard template library).

The main function should next attempt to open and read from another text file, whose name should again be supplied by the user. Each line of this file will contain a module number (which will be a string, e.g. `CE221`), a registration number, and a mark for the module (which will be a real number in the range 0.0 to 100.0). Having input each line you should search the collection for the student object with that registration number and add the module and mark to the map in that object. If the registration number does not match any of the students in the collection a warning message should be output to the screen. If the student already has a mark for the module this mark should be overwritten.

If either file could not be opened the program should terminate cleanly with an appropriate message.

Sample test files containing data in the correct format will be found in the **`ex1`** folder to assist with testing.

Once input is complete details of all of the students in the collection and their marks should be displayed on the screen. A full list of module names and marks should be output for each student, so use of the `<<` operator (which should output only the minimum, maximum and average marks) is not appropriate.

The next requirement is to write, in the same file as the `main` function, two additional functions. The first should have two arguments, a collection of objects of type `Student` (vector, list or set) and a mark (of type `float`) and should print, using the `<<` operator, the names and minimum, maximum and average marks of all students in the collection who have a maximum mark that is greater than or equal to the second argument. The output should be sorted by registration number. If no such students are found an appropriate message should be displayed.

The second function should have three arguments, a collection of students, a module number (of type `string`) and a mark, and should print on the screen the name and the mark for that module of each student in the collection who has a mark for that module that is less than or equal to the third argument . Once again if no such students are found an appropriate message should be displayed.

Finally add to the end of the main program a loop that allows the two functions to be tested interactively. Within the loop body the user should be given three options: test the first function, test the second function or quit. (The prompt for input should make it clear exactly what the user must type to make his or her choice; to facilitate quick testing the expected input options should each be single characters.) If the user selects one of the function-testing options the program should ask for a mark, and a module number if appropriate, and call the chosen function using the collection and the value(s) supplied by the user as arguments.

Exercise 2

In the folder **ex2** in the zip file you will find a `ReadWords` class similar to that used in assignment 1. It has member functions `getNextWord` (which returns the next word from the file whose name was supplied as an argument to its constructor) and `isNextWord` (which returns a boolean value indicating if there is another word in the file). The latter function has been declared as protected, so it can be accessed only in the member functions of the class and its subclasses. The `getNextWord` function calls a private function called `fix` to remove all leading and trailing punctuation from a word; hence it may return an empty string if the "word" contains nothing but punctuation. In this assignment a "word" does not have to contain a letter and no case-conversion is to be performed.

There is also a header file for an abstract class called `ReadFilteredWords`, to be used a base class for developing three derived classes. This is declared as a subclass of `ReadWords` and contains a pure virtual function called `filter` which is intended to be used to select which words should be accepted and which should be rejected using some criterion. Implementations of this in the derived classes should take a word (as returned by `getNextWord`) as an argument and return `true` if the word satisfies the criterion. The class declaration contains a function called `getNextFilteredWord` – this should return the next word from the input (obtained by calling `getNextWord`) that satisfies the criterion, or an empty string if there are no more words that satisfy it. (It is assumed that an empty string will never satisfy a filter, so it cannot be returned if there are more words that satisfy the filter.) You must implement this function in a new **`ReadFilteredWords.cpp`** file.

Three derived classes should be written, each with a different version of `filter` to override the virtual function.

In the first derived class the `filter` function should return `true` if its argument is a string that contains at least one lower-case letter and no upper-case letters. In the second class the `filter` function should return `true` if its argument is a string containing at least twos letter and at least one numeric digit. In the third class the `filter` function should return `true` if its argument is a string that contains at least one letter and at least one punctuation character (as defined by the function `ispunct` declared in the header file **`<cctype>`**).

The three derived classes must be written in separate **`.cpp`** files, each with an associated **`.h`** file.

In a `main` function in a separate file write code that will allow the user to specify a file name and select which derived class should be used, then create an object of that class using the supplied file name as the argument to its constructor. The program should then process the whole file, repeatedly calling `getNextFilteredWord` until an empty string is returned, keeping track the number of occurrences of each filtered word returned by the function using a map.

You must not write three separate copies of code for the three derived classes; you must make use of the virtual function so you will have to use a pointer an object of type `ReadFilteredWords`.

After the processing of the file is complete the program should output the total number of occurrences of filtered words, the number of distinct filtered words (i.e. the number of entries in the map) and the three words with the largest occurrence counts (along with their counts). If there is more than one equally third-most frequent word you should output all of them, unless there are more than 10 in which case you may, if you wish, output the first ten followed by something like "and 15 more words".

It is suggested that you use the Hamlet script supplied for assignment 1 to test your program.

Deliverables

You should submit to FASER a single zip file containing two sub-folders (one for each exercise). These folders must contain all of the .cpp and .h files for the exercise (including any supplied files that you have not modified); it is not necessary to submit executable versions. The only file formats that are acceptable are `.zip`, `.7z`, `.rar` or a linux gzipped tar file.

Marking Scheme

Exercise 1: 15% of the marks for the assignment will be awarded for the `Student` class, 10% for the correct input and display of the collection of students and their marks, 20% for the two functions that search the collections and 5% for the interactive part of the main program.

Exercise 2: 15% will be available for the `getNextFilteredWord` function and the filter classes and 25% for the remainder of the exercise.

The remaining 10% will be awarded for programming style and comments. Your comments should say precisely what each function does; within function bodies you should use only brief comments to say what groups of line do – you must ***not*** say what every statement does. The header files for the filter classes should specify what the filters check for.

[Note that the maximum mark for style and comments will be proportional to the amount of the assignment for which a reasonable implementation attempt has been made; for example if you attempt only exercise 1 you will not earn more than 5.5% of the 10% available for style and comments.]