Programming Assignment: Text File Analyzer

Learning Outcomes

- To gain experience with C++ file input/output techniques
- To gain experience with formatting output
- To practice overall problem-solving skills, as well as general design of a program

Task

In most standard English text, certain letters typically appear with a higher frequency than others. This kind of information has been used in designing games like Scrabble [letter frequencies and point values, etc], as well as more serious applications like decoding encrypted messages and ciphers.

The task is to write a function `q` with prototype

```
void q(char const *input_filename, char const *analysis_file);
```

that reads a text file specified by parameter `input_filename` and prints some statistical results about its contents to an output file specified by parameter `analysis_file`. The results will be primarily an analysis of the characters in the input file, which will include the frequency of occurrence of certain categories of characters, as well as the frequency of occurrences of each of the letters of the alphabet.

The details and requirements of function `q` are:

1. If input file input.txt cannot be opened, the function must print the following error message to standard output:

   ```
   File input.txt not found.

   ```

   If output file output.txt cannot be created, the function must print the following error message to standard output:

   ```
   Unable to create output file output.txt.

   ```

2. Read contents of the input file and print to the output file the following information about the contents of the input file [the following details will be easier to understand if you open for reference an input file such as file1.txt and corresponding output file file1-stats.txt]:
   - A general header and the input file's name.
   - The total number of characters contained in the file.

- A table [with headings] where each row lists a category, the number of occurrences of that category of character, and the percentage of the total characters [in the file] this character category makes up. These are the categories: Letters, White space, Digits, Other characters. Note that the percentages in this chart should add up to 100%. All percentages should be printed to two decimal places, along with a space and a $\%$ sign afterwards, like this: $12.45\ \%$.
- Another heading LETTER STATISTICS indicating that statistics about Latin characters in the input file will be presented. Another table again, with headings [see an output file for details], listing a category, the number of occurrences of that letter type, and the percentage of occurrence of that letter. This time, there are 28 categories: uppercase letters, lowercase letters, and then each of the individual [combining both lower and upper case] Latin letters. Note that the percentage of uppercase and lowercase letters should add up to 100%. Also, the percentages of the 26 alphabet letters should add up to 100%. Specifically, note that these are percentages of the total number of letters, NOT the total number of characters in the file. All percentages should be printed to two decimal places, along with a space and a $\%$ sign afterwards, like this: $12.45\ \%$.
- Carefully examine the input and output files. Your headings and category labels should match mine. *Category labels on your chart should be left aligned. All numbers in your charts [up to this point] should be right aligned*.
- Another heading NUMBER ANALYSIS, followed by the count, sum, and average [to 2 decimal places] of all integer numbers appearing in the input file, where a number is defined as any consecutive sequence of digits bounded by non-digits. For example an input file containing the following text:

  ```
  I am 14 years old and it is now 11:15 AM and my IP address is
  123.45.0.204

  ```

  will have seven "integer numbers": 14, 11, 15, 123, 45, 0, and 204. The average of these numbers should be printed to two decimal places, along with a space and a $\%$ sign afterwards, like this: $12.45\ \%$. *Both the category labels and the numbers should be left aligned*.

Implementation Details

Begin your solution by looking at the input and corresponding output files to understand what needs to be done. Pencil an algorithm that takes the input and transforms it to the corresponding output.

After devising an overall algorithm, you're ready to implement the algorithm in C++. The first step would be to just copy the contents of the input file to the output file. Are you able to make an exact copy in an output file of the input file? If not, you might need to research unformatted reads [in contrast to formatted reads using `operator>>`] from the input file stream using the interface available to objects of type `<ifstream>`.

Once you're able to make an exact replica of the input file in an output file, you're ready to categorize individual characters in the input file and also extract integral values from some of these characters. Write small functions that allow you to perform these categorization and extraction tasks. Don't hardcode simple algorithms - instead convert them to functions so that they can be tested thoroughly and then reused repeatedly.

Writing tabular information to the output file with proper alignment so that your output is exactly similar to my output will be the next challenge. You should look at the manipulators presented in `<iomanip>`: `left`, `right`, `setw`, `fixed`, `setprecision`, `setfill`, and so on.

Rather than trying to complete all the tables in one go, try to understand the manipulators listed above using small programs. Once you've a good understanding of the capabilities of these manipulators, trying implementing a single row of the first table to match the intended output. You can then use the knowledge gleaned from printing this row to implement the remaining rows.

All of this means that you should be able to write the solution only using the following three headers in the C++ standard library: `<iostream>`, `<fstream>`, and `<iomanip>`. Functions declared in other C and C++ standard library headers are not necessary and the auto grader will not accept your submission if you use such functions.

Read this again: ***Your implementation can only include headers `<iostream>`, `<fstream>`, and `<iomanip>`. Addition of any other headers [even in comments] and the use of functions not declared in these headers such as `std::fscanf` or just `scanf` will prevent the auto grader from accepting your submission.***

Header file q.hpp

You must provide a declaration of function `q` in namespace `hlp2` in header file q.hpp. It would look like this:

```
#ifndef Q_HPP_
#define Q_HPP_

// Important note: If there any includes in this file, the
// auto grader will not accept the submission.

namespace hlp2 {
  // declare function q here ...
}

#endif
```

Source file q.cpp

You must provide a definition of function `q` in namespace `hlp2` in source file q.cpp. It would look like this:

```
// You must only include the following header files ...
#include <iostream>
#include <iomanip>
#include <fstream>

// Important notes:

// The auto grader will look for exactly the above three includes.
// If there any additional includes, it will not compile your file.
// The auto grader will not accept any functions not declared in
// these three header files [even in comments]!!!
```

```
// You're warned!!!

namespace hlp2 {
  // provide definition of q here ...
}
```

Driver q-driver.cpp

The driver accepts two additional parameters in addition to the name of the program indicating the name of the input file to analyze and the name of the output file containing the analysis. You are given three files to test: file1.txt, file2.txt, and file3.txt, and corresponding correct output files file?-stats.txt. The auto grader will perform additional tests that are not shown to you - this is done to encourage students to test their code rigorously.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Files to submit

Submit q.hpp and q.cpp.

Compiling, executing, and testing

Download q-driver.cpp, and input and output files from the assignment web page. Create the executable program by compiling and linking directly on the command line:

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror q.cpp q-driver.cpp -o q.out
```

In addition to the program's name, function `main` is authored to take two command-line parameters to specify the name of the input file and the name of the output file.

```
$ ./q.out file1.txt your-file1-stats.txt
```

Compare your submission's output with the correct output in file1-stats.txt:

```
$ diff -y --strip-trailing-cr --suppress-common-lines your-file1-stats.txt file1-stats.txt
```

If the diff command is not silent, your output is incorrect and must be debugged.

You'll have to repeat this process for the remaining input files.

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of g++ options.
    - F grade if your submission doesn't link to create an executable.
    - Your implementation's output doesn't match correct output of the grader [you can see the inputs and outputs of the auto grader's tests]. The auto grader will assign 50% of the grade based on the input and output files given to you. The remaining 50% of the grade will be awarded based on the additional tests implemented by the auto grader.
    - The auto grader will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.
    - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.