CS 220 - Homework 5

Due: Thursday, November 1st by 11pm
This assignment is worth 60 points.

Learning Objectives

This homework will give you practice with C++ STL containers, the string class, file I/O, command-line arguments and input validation. You will use git for version control.

Overview

In this assignment, you write a program to analyze digraphs in an input text file. A digraph is a combination of letters that form one sound in a word (e.g. *ch* in character or *sch* in schindler). The format of an input text file is as follows:

There is an integer number ***n*** (greater than zero) at the beginning of the file showing how many digraphs we have: a list of ***n*** digraphs will follow then. After the list of digraphs, there is a text. A text is a list of words with only a limited set of punctuations. The possible punctuations are period, comma, exclamation mark and question mark. You can assume there will be no other punctuation in the text.

Example of a valid input file (input.txt):

```
5 ch ou ee sch wh

I was lucky. All of a sudden I thought of something that helped make me know I
was getting the hell out. I suddenly remembered this time in around October
that I and Robert Tichener and Paul Campbell were chucking a football around in
front of the academic building. They were nice guys, especially Tichener. It
was just before dinner and it was getting pretty dark out but we kept chucking
the ball around anyway. It kept getting darker and darker and we could hardly
see the ball any more, but we did not want to stop doing what we were doing.
Finally we had to. This teacher that taught biology Mr. Zambesis stuck his head
out of this window in the academic building and told us to go back to the dorm
and get ready for dinner. If I get a chance to remember that kind of stuff I
can get a goodby when I need one, at least most of the time I can. As soon as I
got it, I turned around and started running down the other side of the hill
toward old Spencer house. He did not live on the campus. He lived on Anthony
Wayne Avenue.
```

The program should count how many times each of the provided digraphs occur in the text, case-insensitively. Then, it should print out the list of all the digraphs and containing words (in order of their appearance in the text) in lower case along with their number of occurrences, with the digraphs sorted in one of the three ways specified as a command-line argument. The possible arguments are: 'a' (ASCII order), 'r' (reverse ASCII order), 'c' (count, ordered from largest to smallest, with ties broken by ASCII order).

Example:

```
./digraphAnalyzer input.txt r
```

will run your executable named *digraphAnalyzer* on the input file named *input.txt* (given above) located in the current folder and outputs the digraphs in reverse-ASCII order (e.g. digraph *oo* would be printed before digraph *ea*). Note the punctuations are removed from the words and all output is lower-case only. Possible punctuations are comma, exclamation mark, question mark and period. This will be the output of the above command:

```
wh: [what, when]
sch: []
ou: [thought, out, around, around, out, around, could, out, around,
house]
ee: [see, need]
ch: [tichener, chucking, tichener, chucking, teacher, chance]
q?>
```

As can be seen, the program then awaits the user to enter queries by prompting “q?>”. The user can input 1) a number, 2) a digraph, or) the word “quit”. If a number is entered, it should list all the digraphs (in ASCII order) that occur that many times and their corresponding containing words (in order of their appearance in the text), or print "None" if none exists. If a digraph is entered, it should list how many times the digraph occurs and in which words (in order of their appearance in the text) or "No such digraph" if the digraph is not in the list of input digraphs; 0 would be printed for digraphs that are among the list input digraphs but not found in any word. The program terminates when the word “quit” is typed in.

Sample query runs on the input.txt example:

```
q?>0
sch
q?>2
ee
see
need
wh
what
```

```
when
q?>4
None
q>CH
6
tichener
chucking
tichener
chucking
teacher
chance
q?>sch
0
q?>th
No such digraph
q?>quit
```

Important Note: The program must use container classes from the C++ Standard Template Library (STL) to keep track of digraphs, words, and counts. You must at least use `std::string`, `std::vector` and `std::map`, but you are free to use others as well. Take the time to understand the STL containers; selecting the right ones will make your code cleaner and easier to write and debug.

Special Cases:

- If a certain digraph is contained more than once in a word, count that appearance of the word only once. For example, the digraph *ch* is in the word *chacha* twice, but should only be counted once.
- If a digraph is found in a word that occurs more than once in the text, that word should be counted as many times as it occurs in the text.

Makefile & Compiling

Create and submit a Makefile with a rule that creates a target executable called `digraphAnalyzer`. That target should compile with no errors or warnings using the typical C++ compilation command: `g++ <source> -Wall -Wextra -std=c++11 -pedantic`. Do not use any special libraries that require different additional/different compiler options.

The autograder does an automatic check to see if it can compile this target. ***If your submission fails this test, your final score for the homework will be 0.***

Git log

In the assignments folder of your private repository (*not* the team repository you made for the midterm project), create a new subfolder named `hw5`. Do your work in that subfolder and use `git add`, `git commit` and `git push` regularly. Use `git commit` and the associated comments to document your work. e.g. if you just modified `hw5.cpp` to handle the the "case insensitivity" requirement, you might do `git add hw5.cpp ; git commit -m "Added case insensitivity" ; git push`.

Your submission includes a copy of the output of `git log` showing at least four commits to the repository. Save the `git log` output into a file called `gitlog.txt` (e.g. by doing `git log > gitlog.txt`).

README

Please submit a file called `README` (not `README.txt` or `README.md`, etc -- just `README`) including information about what design choices you made and anything the graders should know about your submission. In `README` you should:

- Write your name and JHED ID at the top of the file
- Briefly justify the structure of your program; why you defined the functions you did, etc
- If applicable: Highlight anything you did that was particularly clever
- If applicable: Tell the graders if you couldn't do everything. Where did you stop? What did you get stuck on? What are the parts you already know do not work according to the requirements?

Specific Requirements

- Your program must have all the functionality described above.
- You must use `std::cin`, `std::cout`, the insertion operator `<<` and the extraction operator `>>` for all input and output. Don't use `printf`, `scanf` or other C I/O functions.
- Your program should not directly allocate or deallocate any heap memory. That is, you should not directly call `malloc`, `free`, `realloc`, `calloc`, `new`, `delete` or similar functions. STL containers or other parts of the C++ library might call these functions, which is fine.
- Your solution should run very quickly, even on large input texts. If your solution is running slowly (more than a couple seconds) think again about whether you are making good use of iterators and the STL containers.
- Factor your code into functions, each function performing a distinct task. Don't do everything in `main`.
- All variables must be declared inside functions. No variables should be global or `extern`.
- Do not use `auto`.

- Use header guards in all header (`.h`) files.
- Do not use `using` in header (`.h`) files.
- In C++ source files (`.cpp` files), you may import individual symbols using statements like "`using std::string`". *Do not* use "`using namespace <id>`", either in headers or in source files.

Hints and Suggestions

Style matters

- ➢ Decompose the entire task into a number of functions with well-defined objectives
- ➢ Comment clearly

Write tests and test your work thoroughly, but you don't have to turn your tests in.

Make use of **gdb** to debug and also run **valgrind** to make sure there is no memory leakage.