CS 215 - Fundamentals of Programming II
Spring 2020 – Extra Credit Project
20 points

Out: April 27, 2020
Due: May 4, 2020 (Monday of finals week, NO LATE WORK ACCEPTED)

This project consists of writing two short applications, one that uses STL vectors and STL lists, and one that uses STL stack. **This assignment is distributed in two separate GitKeeper git repositories**, one for each program. Each program will be graded separately.

Both programs require proper error checking of command-line arguments and file opens. Both programs require the submission of test data files. Neither of the programs require a makefile.

Part I: Creating a concordance

(10 points) Since a list must be searched sequentially, when a list grows to be large, finding a value in the list is not efficient. One way to improve efficiency is to maintain several smaller lists. For example, a text concordance is a list of all distinct words appearing in a particular text. Write a program in the file `concordance.cpp` that reads words in lowercase letters separated by whitespace (space, tab, or newline) and produces a text concordance. (We will ignore uppercase letters, numerals, and punctuation for this program, and it should be assumed that the input consists of only lowercase letters.) The program must meet the following criteria:

- The program must take two command-line arguments, an input file name and an output file name. The words are to be read in from the input file. (Since we are reading words, this should be done using operator>>.)

- The concordance must be implemented as an STL vector with 26 elements that are STL lists of strings. All words beginning with the letter `a` should be stored in the first list in the vector (i.e., the list at index 0), those beginning with `b` should be stored in the second list in the vector (index 1), and so on. Note: you should **compute** the appropriate index. A multi-branch if-statement or a switch statement will not earn full credit. If you do not know how to do this, please ask in class.

- Each word must be inserted into the appropriate list using a function. **This function must receive and pass back a (single) STL list and receive the word to insert.** The result of this function is the word is inserted into the list such that the list is in order based on operator< without duplicates. (I.e., if the item is already in the list, it is ignored.) There are several ways to do this and you are free to do this in any way you see fit as long as it does not involved other data structures. (E.g., you cannot put all of the elements of a list into a vector and call a sorting routine and then put them back into the list.)

- An alphabetic list of the distinct words in the text must be written to the output file, one on each line. The concordance must be written **directly from the (vector of) string list data structures using list iterators**.

- Feel free to use additional functions.

You will write your own test files. **You must submit 1 non-trivial test file.** By non-trivial, we mean a test file of at least several lines of multiple words, with words beginning with a variety of letters, and with duplicates. Please note that there should be no punctuation, no digits, and no uppercase letters. Name this file `sampletext.txt`.

Part II: Checking for matched delimiters

(10 points) Using an STL stack, write a program in the file `match.cpp` that reads (non-whitespace) characters one at a time from a file, and determines if the delimiter pairs `( )`, `{ }`, and `[ ]` in the file match properly. (I.e., it ignores all other characters.) Delimiters match properly if, for each left delimiter, there is a corresponding right delimiter, and the pairs are properly nested. For example,

- `([ ]{( )})` - delimiters match
- `([{ ]( )})` - delimiters do not match, there should be a `}` before the `]`
- `([ ]{( )}` - delimiters do not match, the first `(` has no match at the end
- `[ ]{( )})` - delimiters do not match, the last `)` is extra

The output from this program is to be **written to the screen** (`cout`) and be **exactly one of**:

- `All delimiters match.`

  When the delimiters in the input file match up properly as in the first example.

- `Mismatched delimiter d1 found. Expecting delimiter d2.`

  Where d1 is the current delimiter being considered and d2 is the delimiter that is expected. For the second example above, the mismatch occurs when the `]` is read and a `}` is expected (to match the preceding `{`), so the output for this example would be:

  `Mismatched delimiter ] found. Expecting delimiter }.`

  (**Note** there is one space after the first period.) The program should return 0 immediately when this occurs.

- `Extra delimiter d found.`

  This situation occurs in the third and fourth examples above. Note that d could be either a left or right delimiter. For the third example,the output would be:

  `Extra delimiter ( found.`

  For the fourth example, the output would be:

  `Extra delimiter ) found.`

  The program should return 0 immediately when either of these situations occur.

This program must take one command line argument, the input file name. You will write your own test files. **You must submit 2 non-trivial test files**, one where the delimiters match and one where the delimiters do not

match. By non-trivial, we mean a test file larger than the examples above, including characters other than the delimiters, and of more than one line. Name them **`matched.txt`** and **`unmatched.txt`**, respectively. One input test file you might try is a very simple C++ source file without comments or string literals. (Properly written C++ code should have matched delimiters. However, it is possible to have unmatched delimiters in comments and string literals.)

REMINDER: Your programs must compile for them to be graded. Submissions that do not compile will not graded. Submissions that do not substantially work also will not be graded. The programs will be graded separately, so you should submit each part when it is completed.

Follow the guidelines in the *C++ Programming Style Guideline* handout. As stated in the syllabus, part of the grade on a programming project depends on how well you adhere to the guidelines. The grader will look at your code listing and grade it according to the guidelines.

How to submit

The autotest program requires that the format of any created files or output be exactly as expected including whitespace, capitalization, and spelling. Submit projects by first adding and committing changes, then pushing each project repository. See handout *Using GitKeeper for Submitting Assignments* for additional information.