PA: Using STL to solve real-world problems

Learning Outcomes

- Practice using C++ STL containers [`std::string` and `std::vector`], iterators, and algorithms
- Practice file I/O
- Gain practice in reading and understanding code.

Task

The first task is to define a spell-checker class that will be used to check words for correct spelling. This is a partial definition of class `hlp2::spell_checker` [that you must define in `spelling.hpp`]:

```
class spell_checker {
public:
  enum SCResult { scrFILE_OK = -1,      // File was opened successfully
                  scrFILE_ERR_OPEN = -2, // File was unable to be opened
                  scrWORD_OK = 1,        // Word found in lexicon
                  scrWORD_BAD = 2        // Word not found in lexicon
                };
  struct lexicon_info {
    size_t shortest; // Shortest word in lexicon
    size_t longest;  // Longest word in lexicon
    size_t count;    // Number of words in lexicon
  };

  // Constructor. A lexicon filename must be supplied to initialize
  // private data member but not open the lexicon!!!
  spell_checker(const std::string &lexicon);
  // Count number of words that start with letter. If file can't be
  // opened, return scrFILE_ERR_OPEN. If successful, return scrFILE_OK.
  // The count is returned in the reference parameter count.
  SCResult words_starting_with(char letter, size_t& count) const;
  // Count the number of words that have length 1 to count and store
  // them in lengths at appropriate index. If the file can't be
  // opened, return scrFILE_ERR_OPEN, otherwise return scrFILE_OK.
  SCResult word_lengths(std::vector<size_t>& lengths, size_t count) const;
  // Return some information about lexicon using reference
  // parameter. If the file can't be opened, return scrFILE_ERR_OPEN,
  // otherwise return scrFILE_OK.
  SCResult get_info(lexicon_info& info) const;
  // Lookup the word in lexicon. If the word was found, return
  // scrWORD_OK. If the word was not found, return scrWORD_BAD. If the
  // lexicon file can't be opened, return scrFILE_ERR_OPEN.
  SCResult spellcheck(std::string const& word) const;
  // Given a string, find words in the lexicon that are composed of
  // letters in the same order. If the lexicon can't be opened,
  // return scrFILE_ERR_OPEN. If successful, return scrFILE_OK.
  SCResult acronym_to_word(std::string const& acronym,
```

```
            std::vector<std::string>& words, size_t maxlen = 0) const;
private:
  std::string dictionary;
};
```

Additional details of these member functions must be obtained by reading the unit tests in the driver, by implementing the function with your possibly partial understanding, comparing your output with the correct output, and then refactoring the code, until your output exactly matches the correct output.

In addition to class `hlp2::spell_checker`, you must define a class `hlp2::string_utils` that declare the following static member functions:

```
// Converts lower-case Latin characters in a string to upper-case.
std::string upper_case(std::string const& str);
// Split a string into words. A word is any character but a space.
// Return the word in a vector.
std::vector<std::string> split(std::string const& words);
```

A few notes:

1. All member functions in class `hlp2::spell_checker` [except the constructor] must open the lexicon. If unable to open the file, these member functions must return `scrFILE_ERR_OPEN`; otherwise `scrFILE_OK` must be returned.
2. There is exactly one word per line in the lexicons. All words end with a newline character which is not to be included in the word. Since you're using classes `std::string` and `std::ifstream`, this should be stripped off for you.
3. Member functions `words_starting_with` and `spellcheck` are *case insensitive* when looking for characters or words. You need to account for this. One way to do this is to make everything uppercase or lowercase before comparing.
4. Do not use dynamic memory anywhere. You don't need them. The Standard Library takes care of memory management for you and this means you can't have any memory leaks which is very good for some of you!!!
5. Can you just read the entire lexicon into memory one time instead of opening it and closing it in each member function? The short answer is ***No***. The longer answer is that one of the goals of this exercise is for you to practice opening and [optionally!] closing files in member functions. It is true that loading the entire lexicon into memory will cause word lookups to be faster, since you don't have to go to the disk for each word. However, for this exercise, this time is negligible. In a real spell-checker, you would read the lexicon into memory, but you wouldn't use neither use an array nor a linked list. You would use some sort of balanced tree or hash table, which would speed up the searches dramatically. This topic is beyond the scope of this exercise. You will learn about more efficient approaches to using lexicons in a course specifically about data structures.
6. Do you've to look through the entire lexicon to find a particular word? No. Since the lexicon is sorted, you will know if a word is present or not before reaching the end. For example, if the word you are looking for is `postak`, you may see these words in a lexicon:

```
.
.
.
postage
postages
        <----- (If the word postak existed, it would be here.)
postal
postally
postals
postamputation
.
.
.
```

`postak` belongs between `postages` and `postal`. Once you reach `postal`, you should stop searching. This is simply because `postal` comes after `postak`, so `postak` can't possibly be in the lexicon. This is why lexicons are sorted! Of course, if the word you are looking for is that last one in the lexicon [such as `zyzzyvas`], then you have no choice but to look at every single word.

How to complete exercise

The main purpose of this exercise is to consolidate your knowledge of the C++ standard library and to practice real-world programming. Rather than providing detailed roadmaps to each aspect of the code that you are required to implement, this exercise will require you to understand each function's behavior by examining the unit tests and driver code in `spelling-driver.cpp`. Everything required in this exercise has been presented in class lectures, previous labs, and in-class live code examples. Begin with the first test in function `test1()`. Get this test to pass before moving to `test2()`, and so on.

Submission Details

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

Submission files

You will be submitting interface file `spelling.hpp` [containing definitions of classes `hlp2::spell_checker` and `hlp2::string_utils`] and implementation file `spelling.cpp` [containing definitions of member functions of classes declared in `spelling.hpp`].

Compiling, executing, and testing

Download driver source file `spelling-driver.cpp` containing [lots of] unit tests to test your implementations of classes `hlp2::spell_checker` and `hlp2::string_utils`], input files [the lexicons], and output files [containing correct outputs of unit tests in the driver]. Refactor a `makefile` from previous exercises to test your program.

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 must exactly match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). There are only two grades possible: $A+$ grade if your output matches correct output of auto grader; otherwise F.
    - A maximum of D grade if Valgrind detects even a single memory leak or error. A teaching assistance will check you submission for such errors.
    - 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.