CSE 232

Programming Project #7

The Problem
Auto-completion and spell checkers are everywhere: word processors, phones, etc. Educators complain that students don't know how to spell because spell checkers are so ubiquitous. Let's write a simple program that does completion and spell checking. It isn't efficient enough to be used in a commercial application, but you'll get the idea of how you might write one.

Word-completion
Word completion is often based on the user's typing history, and the suggestions are usually ranked by how frequently the word is used. Also, many tools will give context-based word completion. For this project, we will be writing a very simple word-completor. We simply use a collection of valid words, and return all possible completions of some prefix.

A good example
Peter Norvig, a very well known computer scientist (https://en.wikipedia.org/wiki/Peter_Norvig) has a very popular blog where, one day, he tackled the problem of writing a computer spelling corrector in Python (https://norvig.com/spell-correct.html on a plane trip in around 3-4 hours). We're going to use some of his ideas and write a spell corrector in the same way.

Making spelling mistakes
We will consider 4 kinds of mistakes one might make in typing a word. Let's use the word `something`
- a **delete** : the typist accidentally forgets to type a letter: `somthing`
- an **insert** : the typist "fat fingers" the keys and adds an extra letter: `somethjing`
- a **transpose** : the typist switches the order of two letters : `somehting`
- a **replacement**: the typist types the wrong letter : `somrthing`

Of course, a typist could make more than one of those kinds of mistakes on a word, but the chances are lower that this occurs. However, we'll at least consider the possibility of two such mistakes in the same word.

Basic Premise
We are going to build functions for those 4 kinds of mistakes. The possibilities they generate could, however, be large. Sticking with the word something, what are the counts for the 4 different errors:

- deletes : 9 (there are 9 letters)
- transpose : 8 (there are 8 pairs)
- replacement : 225, 9 locations where 25 possible other letters could be substituted (not counting the letter already there, 226 if we do count those letters)
- inserts : 260 , 10 locations (including the locations before and after the word) * 26 (all letters of the alphabet)

If we take all the possibilities (9+8+225+260 = 502), then we must evaluate whether any of those 502 might be the intended word. We can go one step further. For ***<u>each of the 502 possibilities</u>*** we can run each one through the 4 mistakes generating a yet much larger set, allowing for the possibility of having two errors in a word. I generated the following numbers for the double deletion error `somthng`

Dels: 7 Reps: 176 Trans: 6 Inserts: 201 1 level possibilities: 390	2nd level Dels: 2905 2nd level Reps: 73490 2nd level Trans: 2534 2nd level Inserts: 83240 2nd level possibilities: 70184

You may notice that thelevel possibilities don't add up. That's because there was repetition in the generation of the possibility words. We ignore duplicates. Nonetheless, that is a lot of possible words, far more than we can present to a user. How to sort that out? We can take those 70,000+ words and see if any of them show up in a big list of English words. If we did that, we would get only 4 positive results which I call the reasonables

Reasonables : `something,soothing,sorting,southing`

A good spell checker would offer up those 4, reasonable, possibilities.

Efficiency Matters
You probably knew this, but efficiency is going to matter a bit here. To go from `somthng` to the 4 reasonable replacements should, practically, occur pretty quickly. We probably can't be faster than 2-3 seconds but it can't be 30 seconds. We have to consider efficiency

To that end we are going to use sets and maps. These are data structures that can do fast search/merge/inserts, all of which we are going to need. We will also pass all the sets/maps as references, including the result set/map. That is, we will pass in a set/map which is to contain the answer. In this way our functions don't have to return a result by copy.

Your Tasks
Complete the Project 7 by writing code for the following functions. Details of type for the functions can be found in `proj7_unctions.h` (provided for you, see details below). The

- `void deletes (const string & word, set<string> & result)`
    - creates all the single letter deletes possible from the argument `word` and places them in the argument `result`.
    - `void` return (meaning no return value.) All deletes are inserted into the `set<string> &result`.

- `void replaces(const string &word, set<string> &result)`
    - creates all the single letter replaces possible from the argument `word` and places them in the argument `result`.
        - We allow the original letters to occur in the replacement set, making it potentially +1 against the size of just the replacement set.
    - `void` return (meaning no return value.) All replacements are placed in the `set<string> &result`.

- `void transposes(const string &word, set<string> &result)`
    - creates all the single letter transpositions possible from the argument `word` and places them in the argument `result`.
    - `void` return (meaning no return value.) All transpositions are placed in the `set<string> &result`.

- `void inserts(const string &word, set<string> &result)`
    - creates all the single letter inserstions possible from the argument `word` and places them in the argument `result`.
    - `void` return (meaning no return value.) All inserstions are placed in the `set<string> &result`.

- `void read_words(string fname, set<string> &result)`
    - reads a file, the file name indicated by `fname`, into `set<string> result`. We provide a file named `words.txt` with about 112,000 words, each word on a separate line. Write the function to read files with multiple words on the same line.
    - each word is converted to lower case before it is added to the set.
    - while I believe that all the words in the provided file are alphabetic, you must only store alphabetic characters. Thus non-alphabetic character are deleted: `Two4Me` ❼ `twome`
    - `void` return (meaning no return value.) All insertions are placed in the `set<string> &result`.

-
  ```
  void find_completions(const string &w,
                        const set<string> &word_list,
                        set<string> &result)
  ```
    - takes in a word 'w' as a string, and a set of valid words.
    - `void return`. All the words in word_list that have 'w' as a prefix are added to `result`. This might be a very large set!

- `void find_corrections(const string &word, set<string> &result)`
    - takes in a word as a string, and calculates the deletes, inserts, transposes, and replaces sets for the word.
    - `void return`. All the dels, reps, trans and ins for each word are added to `result`. This might be a very large set!

- `void find_2step_corrections(const string &word, set<string> &result)`
    - takes in a word as a string, and calculates **2 levels** of the deletes, inserts, transposes, and replaces sets for the word. i.e. for each word in 4 sets, all corrections are applied again.
    - `void return`. All the dels, reps, trans and ins for each word are added to `result`. This might be a very, very large set!

-
  ```
  void find_reasonables(const set<string> & possibles,
                 const set<string> &word_list,
                 set<string> & result)
  ```
    - for every word in `possibles`, check if that word exists in the `word_list`. If it does, it is added to `result`
    - `void` return

Deliverables

1. `proj07_functions.cpp` -- your completion of the functions.
2. You are given `proj07_functions.h` and a list of words in words.txt Remember to include your section, the date, project number and comments.
3. Turn it into Mimir as always. Remember, no main!

Assignment Notes

1. If you are going to use algorithms, and you probably should, you have to be careful about constant parameters. You cannot iterate form a `.begin()` to `.end()` on a `const set<string>&`. What those functions yield are pointers, which could be used to violate constant-ness. However, you can use `.cbegin()` to `.cend()`, which yields constant pointers and that will work.
2. The `set_union` algorithm creates a union of two STL data structures, placing the result in a third data structure. The first 4 arguments are the `.begin()` and `.end()` of the two structures being unioned. However, the 5 argument, for this exercise, should be an `inserter` iterator to a `set`. This allows for inserting to a set and removing a duplicate if it occurs. The `inserter` iterator takes two arguments: the set we are inserting into and the location to do the insertion. The location is really irrelevant as the set will re-order itself anyway, but it must be provided. We use the `.end()` iterator as a place for the location (because it's easy).

```
#include<algorithm>
using std::set_union;
#include<iterator>
using std::inserter;


…
set<long> s1 ={1,2,3,4,5};
set<long> s2 = {3,4,5,6,7};
set<long> result;
set_union(s1.begin(), s1.end(),
          s2.begin(), s2.end(),
          inserter(result, result.end() );
```

…

   See the example `inserter.cpp` in the project directory.

3. You should <u>**never**</u> check for duplicates when you are adding something to a set. The set will <u>**automatically**</u> stop a duplicate from being added. Just insert everything in and let the set sort it out.
4. We pass everything by constant reference except the parameter that will contain the result. We pass that as a reference, not const, so we can modify it. This assumes that you pass the result with the correct values in place (empty perhaps, a union of 2 other sets, whatever is appropriate).
5. To find if something is in a set (such as a word from the possibles in the word_list), you can do three things, one of them bad:
   a. `myset.count(the_word)` ❼ 0 if not there, 1 if it is
   b. `myset.find(the_word)` ❼ `myset.end()` if not there, an iterator to the word if it is
   ~~c. find(myset.begin(), myset.end(), the_word);~~

   The first and the second are fast algorithms. They use the alphabetical order of the set to do a binary search. This is (for lack of a better analogy), like the Price is Right Clock Game (http://priceisright.wikia.com/wiki/Clock_Game). You guess a price, the host says "higher" or "lower" and you guess again. The strategy, to us anyway, is obvious. Guess the middle value. If "lower", then guess in the middle of the lower range, otherwise in the higher range. Repeat.

The last one does a linear search (looking at each element in order). That is much slower!!!

6. Overall, algorithms are going to be the way to solve this. Look at the algorithms we mentioned in the videos and notes and see if they apply. If you have questions on Piazza, questions about algorithms (which to use, which is most appropriate, how to use it) would help you the most.