Ben Gafford, Xinya Yang

Problem
Create a spelling correction function which will take in a body of text, and return the same body of text with corrected spelling.

This problem can be decomposed into two primary subproblems:

- Edit distance calculation
- Spelling correction

We use Levenshtein Distance as the means for quantifying the distance between two words (edit distance).

Given an arbitrary dictionary, we can perform spell checking over a body of text by checking to see if each word is in the dictionary, and fixing those that are not. If a given word is not in the dictionary, then we would compute the edit distance between the incorrect word and all given words in the dictionary. We then identify the word with the lowest edit distance to the incorrect word, and replace accordingly.

Major functions
edit_distance(string s1, string s2):
To calculate edit distance, we leverage dynamic programming. Each cell represents the edit distance of the two substrings up to this index. If the two current characters are the same, we copy the value from the top-left diagonal into the current cell. If the characters are not the same, then we add one to the minimum of the top, left, and top-left cells, and put the result into the current cell.

We initialize the values as following: Each cell in the top row is instantiated with the column index. Each cell in the left column is instantiated with the row index. The first row and first column correspond to the [ε,ε] cell.

Time complexity:
Θ(n*m), where n = len(s1) and m = len(s2).

fix_spelling(string s, unordered_set dict):
This function will first tokenize the input string into words, and then split these words according to punctuation. The non-punctuation part of the word is checked against the dictionary. If the word belongs to the dictionary, this word, along with the punctuation, is added to our clean word list. If the word is not in the dictionary, we go through the dictionary and compute the edit distance between the incorrect word and every word in the dictionary. We then take the word from the dictionary that has the minimum edit distance to the incorrect word, and place this word along with the original punctuation into the cleaned word list.
At the end, we have a cleaned word list, which we can then write to stdout.

Time complexity:
Ω(n), where n is the length of the input string.
O(n*n_l*m*m_l), where n is the number of words in the dictionary, n_l is the maximum length of the dictionary, m is in the , m is the length of the dictionary, and l is the maximum word length in the dictionary.

Instructions to execute code

Compile:
`clang++ -o fix_spelling fix_spelling.cpp`

Run:
`./fix_spelling <example_str>`
The program reads the original string from stdin, and writes the spell-corrected string to stdout.

Example input and output
Your string input: ass 123
Corrected version: Vass 123

Your string input: I want fall breaks!
Corrected version: W want Wall wreaks!

Your string input: this is a yool.
Corrected version: this is a wool.

Your string input: minimun
Corrected version: minimus

Discussion
Practical applications of dynamic programming are interesting, and this project was a good example. We had been familiar with edit distance as a means of fuzzy string matching, and its use as a spell checker was an interesting application.

Something that stood out about this program from an algorithmic point of view was how the common-case (correctly spelled words) in the inputs is drastically more efficient than the uncommon case (incorrectly spelled words). In most applications of a tool like this, performance should be pretty good, but in certain cases with many misspelled words, this could lead to terrible performance.