Hands on Assignment 1

- CSci 2021-10, Fall 2018.
- Released Sept 10, 2018.
- Due Sept 24, 2018 at 11:55 PM

Introduction

Your task for this assignment is to build a command-line spell-checking program. You may be more familiar with spell checking as a feature inside an editor, but what you will build is a separate program that looks through a text file and reports any misspelled words in the file. Additionally, when a word is misspelled, the program should check whether the word is similar to a correct word, and if so suggest the correction. To know what words are correct, the program will take, as another argument, a file containing a list of the legal English words. We'll call this a dictionary file, though it's just the words and not the definitions. To get you started, we've written the parts of the code that read the input files and identify individual words. Your job is to write the rest. In particular, any places where you see **TODO** in a comment are places that you should add code.

There is one unusual feature that the program should have, according to a request we are passing on from the marketing department. In a (perhaps misguided) plan to differentiate the spell checker from its competition and show that it is prepared for the Internet age, this will be the only spell checker that supports the `@` (at) sign in English words. Specifically the `@` sign should be treated as a variation of the lowercase letter `a`. For instance words like `em@il` and `@tmosphere` should be counted as correct, just like the outdated 20th-century spellings `email` and `atmosphere`.

This is the sort of program that you could write in any general-purpose language, and it doesn't depend on any 2021-specific concepts; in fact it might seem like it could have come from CSci 1933. That's on purpose: we've doing this project first to give you some practice writing programs in C in our Linux environment, to help you prepare for later projects where you'll have to work with C while also exploring new concepts.

We ask that you write your program to store the word-list in memory using a **hash table** data structure. Specifically, use what is called a separate-chaining hash table, one in which all the values that hash to the same index are stored in a linked list. This will require an efficient **hash function** to index each word in the dictionary. You will populate the hashtable with words found in the dictionary. Then, you will write a function to check if the words in an arbitrary input file are found in the dictionary. Finally, write a recommender function that suggests the misspelled word's correct spelling.

The program will be called spellcheck and have the following usage:

`spellcheck [-d <debug_flags>] <dictionary_file> <input_file>`.

The angle brackets aren't part of the syntax; they indicate that while `spellcheck` and `-d` are written as is, the arguments described as `dictionary_file` and `input_file` need to be replaced by appropriate file names. The square brackets mean that the `-d` option and its flags argument are optional; but if `-d` is given, the next argument needs to be an integer representing debugging flags.

The program should produce output to the standard output (e.g., using printf) with the following format:

```
<misspelledWord1>: <suggestedWord1>
<misspelledWord2>: <suggestedWord2>
<misspelledWord3>?
...
<misspelledWordN>: <suggestedWordN>
```

In other words, there should be one line per misspelled word. Each incorrect word should appear only once. If your program can find a correct word that is similar to the incorrect word, print it after the incorrect word, separated by a space and a colon. If your program can't find a suggestion, just print a question mark after the incorrect word.

The `dictionary_file` has this specific format:

- There is one word per line, each followed by a newline.
- Words consist only of English letters A-Z or a-z, or apostrophes, and are less than 100 letters.
- Your program should be able to efficiently handle a dictionary on the order of 100,000 lines.
- Words that are capitalized or all-capital-letters in the dictionary represent words that need to be capitalized or all-caps to be correct. For instance "America" not "america", and "USA" not "Usa" or "usa". If multiple capitalizations of a word appear in the dictionary, the ones that require more capital letters will come first (e.g., "Baker" the last name before "baker" the occupation).

The `input_file` should be a file containing ASCII text; beyond that it doesn't have any particular format. Your program should also be able to deal efficiently with a large input file. For instance checking a 10,000 word input file against a 100,000 line dictionary to find 100 misspelled words should require only a fraction of a second.

On the course web site near where you found this handout, you'll be able to download a compressed tar file `ha1.tar.gz` containing the files for this assignment, such as a sample of a dictionary file and some input text files, and a framework `spellcheck.c` containing the code we've already written for you.

Copy the tar file to your home directory and then expand it with this command:

```
tar -xzvf ha1.tar.gz
```

Part A - Data Structure

Your first step in building this program is to construct a data structure that will be used to represent a dictionary.

The dictionary will begin as a text file containing a list of correctly spelled words. Your program should read this text file and insert each word into the hashtable.

If you've taken 1913, 1933, or a similar class, you should already be familiar with how hashtables work. If not, you can look at an old textbook or an web resource like Wikipedia for a review, but remember that such outside sources should only be helping you with concepts: you need to write your own code.

The file I/O is already written and taken care of, so you will only be responsible for:

1. Creating the hashtable.
2. Creating the hash function.
3. Inserting each word in the dictionary into the hashtable.

Your hashtable should:

1. Have an appropriate number of bins to hold around 100,000 entries.
2. Have a well-written hash function that minimizes collisions.
3. Manage collisions properly when they happen.
4. Insert in O(1) time.
5. Find in O(1) expected time. (The time to find an item will likely depend on whether there are collisions. But the time should be O(c) if there are c collisions, and the average number of collisions should be small.)

Avoid having too many collisions in your hashtable. This will slow down the performance of your program and defeats the purpose of using a hashtable. As a rough estimate, if you insert as many entries into the hashtable as it hash buckets, more than half of the buckets should have at least one entry. If only a small fraction or even only one bucket is used, that's a good sign you have a poor hash function. Because the number of words in English is not growing much, it is OK for the number of buckets in your hashtable to be fixed, as long as you choose a large enough fixed value.

While avoiding collisions, it is also necessary that your hash function be compatible with the lookup operations you want to do: words can occur capitalized even if they are not capitalized in the dictionary. So think about what this implies for your hash function.

Part B - Spell Check

The second part of this program is the actual spell-check functionality. Given an input text file, your program will need to read every word in the input file (this part is done for you) and look it up in the dictionary. If the word does not exist in the dictionary, either because it is misspelled, or the dictionary is not comprehensive enough, the word should be printed to the console in the format specified above.

Here are some simple rules to follow:

1. If a word doesn't appear in the dictionary, it is spelled wrong.
2. If a word is capitalized in the dictionary, it must be capitalized in the same way to be spelled correctly. To be more specific, any letter that is capitalized in the dictionary must be capitalized. It is OK if even more letters are capitalized, like if part of a document is written IN ALL CAPITAL LETTERS.
3. Otherwise (if the word is lowercase in the dictionary), capitalization does not matter.

One specific corner case you will need to add some code for has to do with the `'` character, which can represent either a single quote or an apostrophe, depending on where it appears. We want to count an apostrophe as part of a word, since the correct placement of apostrophes is an important part of the spelling of contractions, for instance. However single quotes shouldn't be part of a word. The word recognition code that we have written for you always treats the `'` character as part of a word; but if a word appears in single quotes without whitespace like `'this'`, the word your program should look up in the dictionary is `this`, not `'this'`.

It is recommended to test as many edge cases as you can think of. This program should be robust and handle many different type of spelling mistakes.

Use the provided dictionary as a reference for your tests. This will be the main list of words used in grading, and it is indicative of the size and format of dictionary your program should support well.

Part C - Word Suggestion

The final part of any good spell check software is to suggest possible correct words. You are asked to implement a simple version of this feature that outputs one reasonable suggestion. The suggestion must be a word that already exists in the dictionary. We are defining a 'reasonable suggestion' as a new word that differs from the misspelled word by one character. This includes:

1. Having one extra character somewhere in the word.
2. Having one fewer character somewhere in the word.
3. Having one substituted character somewhere in the word.

For example, suppose a word in the input text was *fumd*. This is obviously misspelled and should be caught by the spell check. One reasonable suggestion would be the word *fund* since there is one substituted character. However, an unreasonable suggestion would be the word *stop* since it has nothing in common with the original word.

Your reasonable suggestion should be printed next to the misspelled word in the output. See the output structure above for more detail about the expected output of your program.

In many cases, there may exist more than one reasonable suggestion. Depending on how this portion of your program is written, it may find one replacement before another. You are only required to output one such suggestion.

If no suggestions are found, print a question mark (?) after the misspelled word as specified above.

Testing

The dictionary that will be used most often for grading is provided (the file is called *dictionary*). Make sure that you have tested your program with this dictionary at least a few times. We also suggest that you create your own, smaller dictionaries to use while developing your hashtable and testing. It will be easier to debug 10 or 100 elements than 100,000.

The same applies for the input file. Try to think of corner cases and test these with different inputs. The inputs you will be graded against will be very large (some including tens of thousands of words), so do make sure your structures and algorithms will scale to that size.

A mechanism for setting the debug level of printouts is built into the program. Feel free to add your own debug levels to help you in your testing. The debug level is an integer, so you can use different values of the variable to select different extra outputs or other debugging code.

Expectations

This project must be done individually. It is not a lab activity, and it is meant to test your C programming abilities, not the skill of your friend or the code written by other people and posted online. That being said, feel free to consult the TAs if you need more clarification or guidance on the assignment. You can find a list of the TA office hours on the course website under the *Staff* tab.

We also expect that you begin this assignment early. All of the Hands-on-Assignments in this class are designed to take between 10 and 20 hours to complete. One sure fire way to do poorly on these assignments is to start at the

last minute and not have enough time to fully think through a solution or ask any questions that you have.

Grading

The project will be graded in two ways.

1. Automated grading scripts to test the accuracy of the output and the efficiency of the program.
2. Manual grading by TAs reading your code.

The grade breakdown will be:

- Data structure and populating the dictionary: 30%
- Correctly identifying misspelled words: 20%
- Reasonable word suggestion: 30%
- Code clarity, style, and comments: 20%

Note

A portion of the grading will be automated, so it is important that your program compiles correctly as submitted, takes its inputs in the format specified above, and produces its output in the format specified above. You will get only partial credit if you have the right concepts, but your program does not do what it is supposed to do.

Submitting

This assignment is due on Monday, Sept 24 at 11:55 pm. Please review the course syllabus for the late submission policy. If you have any questions about the submission process or deadline, please email the TAs at **csci2021f18-010-help@umn.edu**

Please submit your solution to this lab on Moodle. We ask that you only submit one .c file that contains all of your code. Please name the file:

`spellcheck_<X500>.c`

Where `<X500>` is the part of your UMN email address before the `@umn.edu`, also the same as your CSE Labs username and the sometimes called an Internet ID or X.500 identifier.

For example, if your email address is `goldy001@umn.edu`, your filename should look like:

`spellcheck_goldy001.c`