Table of Contents

Speller

tl;dr

Implement a program that spell-checks a file, per the below.

```
$ ./speller texts/lalaland.txt
MISSPELLED WORDS

[...]
AHHHHHHHHHHHHHHHHHHHHHHHHHHHT
[...]
Shangri
[...]
fianc
[...]
Sebastian's
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

Distribution

Downloading

```
$ wget http://cdn.cs50.net/2017/fall/psets/5/speller.zip (http://cdn.cs50.net/2017/
$ unzip speller.zip
$ rm speller.zip
$ cd speller
$ ls
dictionaries/  dictionary.c  dictionary.h  keys/  Makefile  README.md speller.c  te
```

Understanding

Theoretically, on input of size *n*, an algorithm with a running time of *n* is asymptotically equivalent, in terms of *O*, to an algorithm with a running time of 2*n*. In the real world, though, the fact of the matter is that the latter feels twice as slow as the former.

The challenge ahead of you is to implement the fastest spell checker you can! By "fastest," though, we're talking actual, real-world, noticeable time—none of that asymptotic stuff this time.

In `speller.c`, we've put together a program that's designed to spell-check a file after loading a dictionary of words from disk into memory. Unfortunately, we didn't quite get around to implementing the loading part. Or the checking part. Both (and a bit more) we leave to you! But first, a tour.

`dictionary.{c,h}`

Open up `dictionary.h`. Declared in that file are four functions; take note of what each should do. Now open up `dictionary.c`. Notice that we've implemented those four functions, but only barely, just enough for this code to compile. Your job, ultimately, is to re-implement those functions as cleverly as possible so that this spell checker works as advertised. And fast!

`Makefile`

Recall that `make` automates compilation of your code so that you don't have to execute `clang` manually along with a whole bunch of switches. However, as your programs grow in size, make won't be able to infer from context anymore how to compile your code; you'll need to start telling make how to compile your program, particularly when they involve multiple source (i.e., `.c`) files, as in the case of this problem. And so we'll utilize a `Makefile`, a configuration file that tells make exactly what to do. Open up `Makefile`, and let's take a tour of its lines.

The line below defines a variable called `CC` that specifies that make should use `clang` for compiling.

```
CC = clang
```

The line below defines a variable called `CFLAGS` that specifies, in turn, that `clang` should use some flags, most of which should look familiar.

```
CFLAGS = -fsanitize=integer -fsanitize=undefined -ggdb3 -O0 -Qunused-arguments -stc
```

The line below defines a variable called `EXE`, the value of which will be our program's name.

```
EXE = speller
```

The line below defines a variable called `HDRS`, the value of which is a space-separated list of header files used by `speller`.

```
HDRS = dictionary.h
```

The line below defines a variable called `LIBS`, the value of which is should be a space-separated list of libraries, each of which should be prefixed with `-l`. (Recall our use of `-lcs50` earlier this term.) Odds are you won't need to enumerate any libraries for this problem, but we've included the variable just in case.

```
LIBS =
```

The line below defines a variable called `SRCS`, the value of which is a space-separated list of C files that will collectively implement `speller`.

```
SRCS = speller.c dictionary.c
```

The line below defines a variable called `OBJS`, the value of which is identical to that of `SRCS`, except that each file's extension is not `.c` but `.o`.

```
OBJS = $(SRCS:.c=.o)
```

The lines below define a "target" using these variables that tells make how to compile `speller`.

```
$(EXE): $(OBJS) Makefile
	$(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
```

The line below specifies that our `.o` files all "depend on" `dictionary.h` and `Makefile` so that changes to either induce recompilation of the former when you run `make`.

```
$(OBJS): $(HDRS) Makefile
```

Finally, the lines below define another target for cleaning up this problem's directory.

```
clean:
	rm -f core $(EXE) *.o
```

Know that you're welcome to modify this `Makefile` as you see fit. In fact, you should if you create any `.c` or `.h` files of your own. But be sure not to change any tabs (i.e., `\t`) to spaces, since `make` expects the former to be present below each target.

The net effect of all these lines is that you can compile `speller` with a single command, even though it comprises quite a few files:

```
make speller
```

Even better, you can also just execute:

```
make
```

And if you ever want to delete speller plus any `core` or `.o` files, you can do so with a single command:

```
make clean
```

In general, though, anytime you want to compile your code for this problem, it should suffice to run:

```
make
```

`speller.c`

Okay, next open up `speller.c` and spend some time looking over the code and comments therein. You won't need to change anything in this file, but you should understand it nonetheless. Notice how, by way of `getrusage`, we'll be "benchmarking" (i.e., timing the execution of) your implementations of `check`, `load`, `size`, and `unload`. Also notice how we go about passing `check`, word by word, the contents of some file to be spell-checked. Ultimately, we report each misspelling in that file along with a bunch of statistics.

Notice, incidentally, that we have defined the usage of `speller` to be

```
Usage: speller [dictionary] text
```

where `dictionary` is assumed to be a file containing a list of lowercase words, one per line, and `text` is a file to be spell-checked. As the brackets suggest, provision of `dictionary` is optional; if this argument is omitted, `speller` will use `dictionaries/large` by default. In other words, running

```
./speller text
```

will be equivalent to running

```
./speller dictionaries/large text
```

where `text` is the file you wish to spell-check. Suffice it to say, the former is easier to type! (Of course, `speller` will not be able to load any dictionaries until you implement `load` in `dictionary.c`! Until then, you'll see **Could not load**.)

Within the default dictionary, mind you, are 143,091 words, all of which must be loaded into memory! In fact, take a peek at that file to get a sense of its structure and size. Notice that every word in that file appears in lowercase (even, for simplicity, proper nouns and acronyms). From top to bottom, the file is sorted lexicographically, with only one word per line (each of which ends with `\n`). No word is longer than 45 characters, and no word appears more than once. During development, you may find it helpful to provide `speller` with a `dictionary` of your own that contains far fewer words, lest you struggle to debug an otherwise enormous structure in memory. In `dictionaries/small` is one such dictionary. To use it, execute

```
./speller dictionaries/small text
```

where `text` is the file you wish to spell-check. Don't move on until you're sure you understand how `speller` itself works!

Odds are, you didn't spend enough time looking over `speller.c`. Go back one square and walk yourself through it again!

`texts/`

So that you can test your implementation of `speller`, we've also provided you with a whole bunch of texts, among them the script from *La La Land*, the text of the Affordable Care Act, three million bytes from Tolstoy, some excerpts from *The Federalist Papers* and Shakespeare, the entirety of the King James V Bible and the Koran, and more. So that you know what to expect, open and skim each of those files, all of which are in a directory called `texts` within your `pset5` directory.

Now, as you should know from having read over `speller.c` carefully, the output of `speller`, if executed with, say,

```
./speller texts/lalaland.txt
```

will eventually resemble the below. For now, try executing the staff's solution (using the default dictionary) with the below.

```
~cs50/pset5/speller texts/lalaland.txt
```

Below's some of the output you'll see. For information's sake, we've excerpted some examples of "misspellings." And lest we spoil the fun, we've omitted our own statistics for now.

```
MISSPELLED WORDS

[...]
AHHHHHHHHHHHHHHHHHHHHHHHHHHHT
[...]
Shangri
[...]
fianc
[...]
Sebastian's
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

`TIME IN load` represents the number of seconds that `speller` spends executing your implementation of `load`. `TIME IN check` represents the number of seconds that `speller` spends, in total, executing your implementation of `check`. `TIME IN size` represents the number of seconds that `speller` spends executing your implementation of `size`. `TIME IN unload` represents the number of seconds that `speller` spends executing your implementation of `unload`. `TIME IN TOTAL` is the sum of those four measurements.

Note that these times may vary somewhat across executions of `speller`, depending on what else CS50 IDE is doing, even if you don't change your code.

Incidentally, to be clear, by "misspelled" we simply mean that some word is not in the `dictionary` provided.

Questions

Open up `README.md` and answer each of the questions therein.

Specification

Alright, the challenge now before you is to implement `load`, `check`, `size`, and `unload` as efficiently as possible, in such a way that `TIME IN load`, `TIME IN check`, `TIME IN size`, and `TIME IN unload` are all minimized. To be sure, it's not obvious what it even means to be minimized, inasmuch as these benchmarks will certainly vary as you feed `speller` different values for `dictionary` and for `text`. But therein lies the challenge, if not the fun, of this problem. This problem is your chance to design. Although we invite you to minimize space, your ultimate enemy is time. But before you dive in, some specifications from us.

- You may not alter `speller.c`.
- You may alter `dictionary.c` (and, in fact, must in order to complete the implementations of `load`, `check`, `size`, and `unload`), but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
- You may alter `dictionary.h`, but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
- You may alter `Makefile`.
- You may add functions to `dictionary.c` or to files of your own creation so long as all of your code compiles via `make`.
- Your implementation of `check` must be case-insensitive. In other words, if `foo` is in dictionary, then `check` should return true given any capitalization thereof; none of `foo`, `foO`, `fOo`, `fOO`, `fOO`, `Foo`, `FoO`, `FOo`, and `FOO` should be considered misspelled.
- Capitalization aside, your implementation of `check` should only return `true` for words actually in `dictionary`. Beware hard-coding common words (e.g., `the`), lest we pass your implementation a `dictionary` without those same words. Moreover, the only possessives allowed are those actually in `dictionary`. In other words, even if `foo` is in `dictionary`, `check` should return `false` given `foo’s` if `foo’s` is not also in `dictionary`.
- You may assume that `check` will only be passed strings with alphabetical characters and/or apostrophes.
- You may assume that any `dictionary` passed to your program will be structured exactly like ours, lexicographically sorted from top to bottom with one word per line, each of which ends with `\n`. You may also assume that `dictionary` will contain at least one word, that no word will be longer than `LENGTH` (a constant defined in `dictionary.h`) characters, that no word will appear more than once, and that each word will contain only lowercase alphabetical characters and possibly apostrophes.
- Your spell checker may only take `text` and, optionally, `dictionary` as input. Although you might be inclined (particularly if among those more comfortable) to "pre-process" our default dictionary in order to derive an "ideal hash function" for it, you may not save the output of any such pre-processing to disk in order to load it back into memory on subsequent runs of your spell checker in order to gain an advantage.

- Your spell checker may not leak any memory.
- You may search for (good) hash functions online, so long as you cite the origin of any hash function you integrate into your own code.

Alright, ready to go?

1. Implement `load`.
2. Implement `check`.
3. Implement `size`.
4. Implement `unload`.

Walkthrough

Hints

Be sure to `free` in `unload` any memory that you allocated in `load`! Recall that `valgrind` is your newest best friend. Know that `valgrind` watches for leaks while your program is actually running, so be sure to provide command-line arguments if you want `valgrind` to analyze `speller` while you use a particular `dictionary` and/or text, as in the below. Best to use a small text, though, else `valgrind` could take quite a while to run.

```
valgrind ./speller texts/ralph.txt
```

If you run `valgrind` without specifying a `text` for `speller`, your implementations of `load` and `unload` won't actually get called (and thus analyzed).

If unsure how to interpret the output of `valgrind`, do just ask `help50` for help:

```
help50 valgrind ./speller texts/ralph.txt
```

Testing

How to check whether your program is outting the right misspelled words? Well, you're welcome to consult the "answer keys" that are inside of the `keys` directory that's inside of your `speller` directory. For instance, inside of `keys/lalaland.txt` are all of the words that your program *should* think are misspelled.

You could therefore run your program on some text in one window, as with the below.

```
./speller texts/lalaland.txt
```

And you could then run the staff's solution on the same text in another window, as with the below.

```
~cs50/pset5/speller texts/lalaland.txt
```

And you could then compare the windows visually side by side. That could get tedious quickly, though. So you might instead want to "redirect" your program's output to a file, as with the below.

```
./speller texts/lalaland.txt > student.txt
~cs50/pset5/speller texts/lalaland.txt > staff.txt
```

You can then compare both files side by side in the same window with a program like `diff`, as with the below.

```
diff -y student.txt staff.txt
```

Alternatively, to save time, you could just compare your program's output (assuming you redirected it to, e.g., `student.txt`) against one of the answer keys without running the staff's solution, as with the below.

```
diff -y student.txt keys/lalaland.txt
```

If your program's output matches the staff's, `diff` will output two columns that should be identical except for, perhaps, the running times at the bottom. If the columns differ, though, you'll see a `>` or `|` where they differ. For instance, if you see

```
MISSPELLED WORDS                                               MISSPELLED WORDS

TECHNO                                                         TECHNO
L                                                              L
                                                             > Thelonious
Prius                                                          Prius
                                                             > MIA
L                                                              L
```

that means your program (whose output is on the left) does not think that `Thelonious` or `MIA` is misspelled, even though the staff's output (on the right) does, as is implied by the absence of, say, `Thelonious` in the lefthand column and the presence of `Thelonious` in the righthand column.

`check50`

To test your code less manually (though still not exhaustively), you may also execute the below.

```
check50 cs50/2018/x/speller
```

Note that `check50` will also check for memory leaks, so be sure you've run `valgrind` as well.

Staff's Solution

How to assess just how fast (and correct) your code is? Well, as always, feel free to play with the staff's solution, as with the below, and compare its numbers against yours.

```
~cs50/pset5/speller texts/lalaland.txt
```