Computer Science 50: Introduction to Computer Science I
Harvard College
Fall 2008

Problem Set 6: Mispellings[1]

due by 7:00 P.M. on Friday, 14 November 2008

Goals.

- Allow you to design and implement your own data structure(s).
- Optimize your code's (real-world) running time.
- Challenge THE BIG BOARD.

Recommended Reading.

- Sections 18 – 20, 27 – 30, 33, 36, and 37 of `http://www.howstuffworks.com/c.htm`.
- Chapter 26 of *Absolute Beginner's Guide to C*.
- Chapter 17 of *Programming in C*.

image from `http://img.dailymail.co.uk/i/pix/2007/09_01/misspelledAP0609_468x311.jpg`

[1] Yes, you're very sharp. But we know. It's meant to be ironic. Or something. We do go to the same shcool, you know.

Academic Honesty.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed (*e.g.*, by some problem set or the final project). Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, magazine, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student.

Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the staff.

You may even turn to the Web for instruction beyond the course's lectures and sections, for references, and for solutions to technical difficulties, but not for outright solutions to problems on problem sets or your own final project. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course's lectures and sections (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

All forms of academic dishonesty are dealt with harshly.

Grades.

Your work on this problem set will be evaluated along three primary axes.

Correctness. To what extent is your code consistent with our specifications and free of bugs?
Design. To what extent is your code written well (*i.e.*, clearly, efficiently, elegantly, and/or logically)?
Style. To what extent is your code readable (*i.e.*, commented and indented with variables aptly named)?

Getting Started.

☑ Check!

☐ SSH to `cloud.cs50.net` and execute the command below.

```
cp -a ~cs50/pub/distributions/pset6/ ~
```

Now pull up the man page for `cp`. Notice that this `-a` switch is defined as follows.

```
-a, --archive
       same as -cdpPR
```

Those lines mean that running `cp` with `-a` (or `--archive`) is the same as running `cp` with `-cdpPR` (or, equivalently, `-c -d -p -P -R`). The last of those switches (`-R`) should be familiar, as it's equivalent to `-r`, which you've used in past problem sets to copy directories recursively. The second-to-last (`-P`) is probably new to you. Per `cp`'s man page, this switch means "never follow symbolic links." A symbolic (also known as a symlink or soft link) is a special kind of file that pretty much just contains the path to another file. In fact, let's take a look at what you just copied. Navigate your way to `~/pset6/` and type the (familiar) command below.

```
ls
```

You should see the below.

```
Makefile  dictionary.c  dictionary.h  questions.txt  speller.c  texts@
```

Notice that `texts` has a trailing `@`. Now execute the (also familar) command below.

```
ls -l
```

You should see output like the below.

```
-rw-r--r-- 1 cs50 cs50  533 Nov  7 19:00 Makefile
-rw-r--r-- 1 cs50 cs50  971 Nov  7 19:00 dictionary.c
-rw-r--r-- 1 cs50 cs50  884 Nov  7 19:00 dictionary.h
-rw-r--r-- 1 cs50 cs50    0 Nov  7 19:00 questions.txt
-r--r--r-- 1 cs50 cs50 5310 Nov  7 19:00 speller.c
lrwxrwxrwx 1 cs50 cs50   32 Nov  7 19:00 texts -> /home/cs50/pub/share/pset6/texts/
```

As you might have guessed, `texts` is a symbolic link. From the looks of the above, it looks like `texts` leads to an actual directory called `texts` in `/home/cs50/pub/share/pset6/`. Let's take a look. Go ahead and execute the below.

```
cd texts/
```

Now execute the below.

```
pwd
```

Recall that `pwd` reports your current working directory. Seems you're now in `/home/cs50/pub/share/pset6/texts/`, even though your prompt (parenthetically) says otherwise! Neat, eh?[2] Of course, executing the below won't bring you back to the directory you were just in (*i.e.*, `~/pset6/`), as it normally does.

```
cd ..
```

Rather, you should find yourself in `/home/cs50/pub/share/pset6/`. That is, after all, the true parent of `/home/cs50/pub/share/pset6/texts/`. So, to get back to your own `~/pset6/` directory, go ahead and execute the below.

```
cd ~/pset6/
```

Why all this talk about symlinks? Well, rather than waste space by giving each of you your own copy of `texts` (inside of which is a whole bunch of read-only files), we decided to put one copy in the cloud but make it easy for you to get at. In other words, because you have a symlink to `/home/cs50/pub/share/pset6/texts/` in your `~/pset6/` directory, you'll be able to access files therein by way of a much shorter relative path. In fact, let's bring this point home. Assuming you're in `~/pset6/`, go ahead and execute the below.[3]

```
ls texts/
```

You should see a whole bunch of (fun) files, as though `texts` were, in fact, a normal directory inside your `~/pset6/` directory. Even though it's not!

So, why was `-a` (and, in turn, `-P`) useful with `cp`? Well, that flag ensured that the symlink to `/home/cs50/pub/share/pset6/texts/` was not "followed" but instead copied as a symlink, else you'd end up with your own copy of that directory (since that's where you end up by following the symlink). As for those other flags to `cp` (`-c`, `-d`, and `-p`), well, `man` is your friend if you're curious as to what they do!

Want to procrastinate (and practice "following" a symlink)? Try out the command below from within `~/pset6/`.[4]

```
more texts/austinpowers.txt
```

Yeah, baby, yeah!

☐ Gentlemen, are we all here? Good. As you know, my plot to high-jack nuclear weapons and hold the world hostage has failed. Again. This organization will not tolerate failure. Mustafa... Frau Farbissina... I spared your lives because I need you to help me rid the world of the only man who can stop me now. We must go to London. I've set a trap for Austin Powers![5]

[2] Remember Chutes and Ladders? This is exactly like that. But not really.
[3] Be sure to include that trailing slash (else you'll just be reminded that `texts` is a symlink)!
[4] Hit `q` to quit out of `more`!
[5] Ha, made you procrastinate.

Alotta Mispellings.

☐ 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 seconds—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 143,091-word dictionary from disk into memory. Unfortunately, we didn't quite get around to implementing the loading part. Or the checking part. Both (and more) we leave to you!

Before we walk you through `speller.c`, go ahead and open up `dictionary.h` with Nano. 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 for this problem set is to re-implement those functions as cleverly as possible so that this spell-checker works as advertised. And fast!

Let's get you started.

☐ Open up `speller.c` with Nano 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

```
speller [dict] file
```

where `dict` is assumed to be a file containing a list of lowercase words, one per line, and `file` is a file to be spell-checked. As the brackets suggest, provision of `dict` is optional; if this argument is omitted, `speller` will use `/home/cs50/pub/share/pset6/dict/words` by default for its dictionary. Within that file are those 143,091 words that you must ultimately load into memory. In fact, take a peek at that file with Nano (or `more` or `less`) 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 `dict` of your own that contains far fewer words, lest you struggle to debug an otherwise enormous structure in memory.

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!

☐ Okay, technically that last problem induced an infinite loop. But we'll assume you broke out of it. In `questions.txt`, answer each of the following questions in one or more sentences.

0. What is pneumonoultramicroscopicsilicovolcanoconiosis?
1. According to its man page, what does `getrusage` do?
2. Per that same man page, how many members are in a variable of type `struct rusage`?
3. Why do you think we pass `before` and `after` by reference (instead of by value) to `calculate`, even though we're not changing their contents?
4. Explain as precisely as possible, in a paragraph or more, how `main` goes about reading words from a file. In other words, convince us that you indeed understand how that function's `for` loop works.
5. Why do you think we used `fgetc` to read each word's characters one at a time rather than use `fscanf` with a format string like `"%s"` to read whole words at a time? Put another way, what problems might arise by relying on `fscanf` alone?

☐ Now take a look at `Makefile`. Notice that we've employed some new tricks. Rather than hard-code specifics in targets, we've instead defined variables (not in the C sense but in a `Makefile` sense). The line below defines a variable called `CC` that specifies that make should use GCC for compiling.

```
CC = gcc
```

The line below defines a variable called `CFLAGS` that specifies, in turn, that GCC should use some familiar flags.

```
CFLAGS = -ggdb -std=c99 –Werror
```

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 `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 default target using these variables.

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

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

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

Finally, the lines below define a target for cleaning up this problem set'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.

Even though this `Makefile` is fancier than past ones, compiling your code remains as easy as executing the below.

```
make
```

But now you know how to make (pun intended) a more sophisticated `Makefile`!

☐ On to the most fun of these files! Notice that in `~/pset6/texts/`, we have provided you with a number of texts with which you'll be able to test your `speller`. Among those files are the script from *Austin Powers: International Man of Mystery*, a sound bite from Ralph Wiggum, three million bytes from Tolstoy, some excerpts from Machiavelli and Shakespeare, the entirety of the King James V Bible, and more. So that you know what to expect, take a peek at those files using `cat`, `less`, or `more`. You can also use Nano, but be sure not to save any changes accidentally.

Now, as you should know from having read over `speller.c` carefully, the output of `speller`, if run on, say, `austinpowers.txt`, should resemble the below. For amusement's sake, we've excerpted some of our favorite "misspellings." And lest we spoil the fun, we've omitted our own statistics for now.

```
MISSPELLED WORDS

[...]
Bigglesworth
[...]
Fembots
[...]
Virtucon
[...]
friggin'
[...]
shagged
[...]
trippy
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN FILE:
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.

Incidentally, to be clear, by "misspelled" we mean that some word is not in the `dict` provided. "Fembots" might very well be in some other (swinging) dictionary.

☐ Alright, the challenge ahead of 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 `dict` and for `file`. But therein lies the challenge, if not the fun, of this problem set. This problem set 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.

i. You may not alter `speller.c`.
ii. 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`.
iii. You may alter `dictionary.h`, but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
iv. You may alter `Makefile`.
v. You may add functions to `dictionary.c` or to files of your own creation so long as all of your code compiles via `make`.
vi. Your implementation of `check` must be case-insensitive. In other words, if `foo` is in `dict`, 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.
vii. Capitalization aside, your implementation of `check` should only return `true` for words actually in `dict`. Beware hard-coding common words (*e.g.*, `the`), lest we pass your implementation a `dict` without those same words. Moreover, the only possessives allowed are those actually in `dict`. In other words, even if `foo` is in `dict`, `check` should return `false` given `foo's` if `foo's` is not also in `dict`.
viii. You may assume that `check` will only be passed strings with alphabetical characters and/or apostrophes.
ix. You may assume that any `dict` 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 no word will be longer than `LENGTH` (a constant defined in `speller.c`), characters, that no word will appear more than once, and that each word will contain only lowercase alphabetical characters, apostrophes, and/or newlines.
x. Your spell-checker may only take `file` and, optionally, `dict` 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.

Alright, ready to go?

☐ Implement `load`!

Allow us to suggest that you whip up some dictionaries smaller than the 143,091-word default with which to test your code during development.

☐ Implement `check`!

Allow us to suggest that you whip up some small files to spell-check before trying out, oh, War and Peace.

☐ Implement `size`!

If you planned ahead, this one is easy!

☐ Implement `unload`!

Be sure to free any memory that you allocated in `load`!

☐ In fact, be sure that your spell-checker doesn't leak any memory at all. Remember 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 `dict` and/or `file`, as in the below.

```
valgrind -v --leak-check=full speller texts/austinpowers.txt
```

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

☐ Don't forget about your other good buddy, GDB.

☐ And the course's bulletin board, who wishes you wrote him more often.

☐ How to assess just how fast (and correct) your code is? Well, feel free to play with the staff's solution, as in the below.

```
~cs50/pub/solutions/pset6/speller texts/austinpowers.txt
```

But also feel free to put your code to the test against your own classmates'! Execute the command below to challenge THE BIG BOARD.

```
bigboard
```

We'll benchmark your spell-checker with a variety of inputs. Assuming your output's correct, you can then surf on over to the course's home page to see how your `speller` stacks up against others'! Feel free to challenge THE BIG BOARD as often as you'd like; it will display your most recent results.[6]

We shall honor those atop THE BIG BOARD.

Realize that, for convenience, THE BIG BOARD includes links to lists of words considered misspellings (with respect to our specifications and that 143,091-word dictionary) for each of the texts in `~/pset6/texts/`.

By the way, you might want to turn off GCC's `-ggdb` flag when challenging THE BIG BOARD. And you might want to read up on GCC's `-O` flags! (Remember how?)

Those more comfortable might also find such tools as `gprof` and `gcov` of interest.

☐ Congrats! At this point, your speller-checker is presumably complete (and fast!), so it's time for a debriefing. In `questions.txt`, answer each of the following questions in a short paragraph.

6. Why do you think we declared so many parameters as being `const` in `dictionary.c` and `dictionary.h`?
7. What data structure(s) did you use to implement your spell-checker? Be sure not to leave your answer at just "hash table," "trie," or the like. Expound on what's inside each of your "nodes."
8. How slow was your code the first time you got it working correctly?
9. What kinds of changes, if any, did you make to your code over the course of the week in order to improve its performance?
10. Do you feel that your code has any bottlenecks that you were not able to chip away at?

[6] Realize, incidentally, that your spell-checker's performance might very well vary based on what others are doing on `cloud.cs50.net` at the moment you challenge. That reality, however, is part of the challenge!

Submitting Your Work.

☐ Ensure that your work is in `~/pset6/`. Submit your work by executing the command below.

```
cs50submit pset6
```

Thereafter, follow any on-screen instructions until you receive visual confirmation of your work's successful submission. You will no longer receive "receipts" via email; `cs50submit`'s own output will confirm or deny your successful submission. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.