HW 4: Debugging

Introduction

In this assignment you will need to finish the implementation of a Dictionary module that can perform look-up operations, and a driver program that uses the module and generates statistics about a given English text.

We will provide you with the majority of the code that already has most of the functionality, but is full of nasty memory errors. And your job is to understand how memory is managed and used in the provided code, identify those errors, and come up with a solution to those errors.

Starter Code & Submitting

You can download the code to klaatu or the VM by running the following command:

```
wget
https://courses.cs.washington.edu/courses/cse374/20au/homeworks/hw4.tar.gz
tar -xzf hw4.tar.gz
```

You will see a folder called hw4_starter_code containing the following files:

- `SpellChecker.h`, contains the declaration of functions in the Dictionary module.
- `SpellChecker.c` contains the implementation of functions in `SpellChecker.h`.
- `Count_typos.c` the driver program.
- `Utils.h` contains the declaration of utility functions.
- `Utils.c` the implementation of utility functions.

The only files that you'll need to modify and submit are

- `SpellChecker.c`
- `count_typos.c`

Although there are only two files that contain the buggy code, you'll still want to look into the header files to see what each function is supposed to do, and to decide how you may modify the code while satisfying the spec.

Getting Started

This homework implements a program that checks spelling of English text and outputs relevant statistics on the number of words, paragraphs, typos, etc.. This program is case insensitive, so it will recognize "Hello" in the input text as "hello" in the dictionary.

The flow of the program, in general, goes like this:

- Build a dictionary from a certain file that contains all the words we want in the dictionary.
- Read and examine the input text word by word, updating variables holding statistics and generating output about specific mistyped words along the way.
- Write the human-readable statistics to the output.

Types

This homework involves usage of some **types** that we'd like to address before you look at the actual code:

- `size_t`
- `char*` and `char**`
- `int` vs. `char`

On `size_t`:

The type `size_t` is equivalent to `unsigned long`, and is conventionally used in C standard library functions to store the size of a given type, as computed by calling `sizeof` on a variable or type name. You can think of the name as "size type." On the klaatu system (64-bit) this will be a 64-bit unsigned integer.

On `char*` and `char**`:

As we're all well aware, the type `char*` is a pointer to a null-terminated array of `char`'s. The naive implementation of our dictionary is simply a `char**`: pointing to an array of `char*`'s, each points to a string containing a single word. And our `check_spelling` operation is then a grass-root **binary search** on the array to see if a word exists. As such, all operations on our dictionary requires the user to provide both the dictionary itself and the size of the dictionary; because we wouldn't know how many `char*`'s the dictionary has or what the bounds of our binary search will be otherwise.

On `int`, `char` (read if you're interested in understanding Utils.c):

Recall that `int` is a 32-bit/4-byte signed integer. `char` may be considered a specialized type for character, but in reality it is the same as `int` except that it's shorter with only 8 bits. In fact, they're completely interchangeable if the value doesn't go past the bounds of an 8-bit space. Suppose that we have:

```
int i = 48;
char c = '0';  // the ascii value of '0' is decimal 48
```

The following statements will both print out "48":

```
printf("%d\n", i); // %d is the placeholder for integer values
                      to be interpreted as decimal numbers
printf("%d\n", c); //
```

And the following will both print out “0”:
```
printf("%c\n", i); // %c is the placeholder for ascii characters
printf("%c\n", c);
```

Also, this statement:
`int i = '0';`
is identical to:
`int i = 48;`
And so are these two:
`char c = '0';`
&
`char c = 48;`

Ultimately, you can even do this:
```
char c = 48, d = 5;
c = c + d; // c is now decimal 53, which is the ascii value of '5'
```

With all of that said, it should be clear that `int` and `char` are **identical** in terms of binary representation and it’s up to the programmer how a specific instance of each is interpreted.

In Utils.c, we chose to use `int` to store the current character as we step down along the `FILE*` stream; this is because we would like to be able to look at the current character and decide if it’s `EOF`; it just so happens that `EOF` is of type `int`.
(Alternatively, we can use `feof()` to detect `EOF` as well)

Directions

Compile-time errors

To start this homework, simply `cd` to the starter code folder and run this command:

```
$ make
```

And you’ll notice many compiler errors that come up.
To address this, you’ll need to add certain `#include` statements to the appropriate places. To find out what should be included in which file, you may refer to man pages, online documentation and the error message from `gcc` for headers in C standard library. For others, consider reading through the provided code, or at least the comments of them, to identify how each of the code files are connected.

Runtime errors

Most of the work for this homework will happen here.
The provided code is designed to be **infested** with all kinds of memory errors that programmers can easily make due to negligence. And your task will be to identify and get rid of them; the major, obvious places where you have to make an addition are marked with `// TODO` but there are many more hidden ones that you'll have to identify with `gdb` or `valgrind`.
To clarify, it is suggested that you remove the TODO comments once you're done with them.

To approach this part of the homework, we have a few tips that we recommend:

- `valgrind` is your absolute best friend for this assignment; however, the starter code as it is, makes many and aggressive memory errors that it can cause `valgrind` to freeze/crash. This means that sometimes you'll have to `ctrl-c` to terminate it and make use of whatever information that is printed on the screen.
- `valgrind` options, such as `--leak-check=full`, `--show-leak-kinds=all`, `--track-origins=yes`, are all very helpful but they do slow down the execution by a certain amount. Consider using them progressively.
- If you have a hard time locating the bug with the information given by `valgrind`, consider stepping through the program to the segment that causes the error using `gdb`, and pay extra attention whenever you see a `malloc` operation or a write to heap memory.

Style errors

It is recommended that you look at the style errors after you're done with all the other ones; because fixing style won't involve any changes to the structure or the behaviors of the program. We didn't deliberately include any style errors for you to fix.

Please use python 2 to run `clint.py`

Goal

In the end, the program needs to satisfy the following requirements:

- The code needs to compile without any warning by running `make`.
- There should be no memory leaks.
- Your program's behaviors need to match **EXACTLY** the behaviors of the `solution_binary` given to you in the starter code.

Code Quality Requirements

This assignment will not be graded on styles, because 95% of the code is written by the staff and there are only a few lines of code that you need to change/add to make it work. However, we do still recommend putting short comments where you do make changes, to

1. Help identify the part you changed
2. Reinforce good coding practice