Exam 2

Intro

You will have 2 hours to complete this computer-based exam. A copy of these instructions will be provided as a printout.

For this exam, you will have to debug C code using `gdb` and `valgrind`.

Before beginning the exam, you should run the following command in exam directory:

```
chmod +x start
./start
```

This will setup the test environment that stores your inputs, which will enable us to give you partial credits.

Specification

This program implements the basic functionality of checking spelling of English text and outputting 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 goes like this:

- Build a dictionary from a certain file that contains all the words we want in the dictionary.
- The input text is read and examined 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.

The provided code 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

File structure

```
source/
├── dict/                   # dictionaries
│   ├── tiny.txt
│   ├── small.txt
│   └── large.txt
├── books/                  # books to check
│   ├── pap.txt
│   ├── scramble.txt
│   └── simple.txt
```

```
├── Makefile                # Makefile
├── solution_binary         # example solution binary for verification
├── compare_to_solution.sh  # verification script
├── count_typos.c           #      <-- Modify this
├── SpellChecker.c          #      <-- Modify this
├── SpellChecker.h          # SpellChecker header
├── Utils.c                 # Utilities, don't modify
└── Utils.h                 # Utilities header
```

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 need to look into the header files to see what each function is designed for, and decide how you may modify the code while satisfying the spec.

Do note that you're able to open multiple windows for easier navigation while fixing the bugs.

Types in C

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

On `size_t`:

The type `size_t` is conventionally used in C standard library functions to store a number big enough to fit the size of any object, as computed by calling `sizeof` on a variable or type name. You can think of the name as "size type." On Linux (64-bit) this will be a 64-bit unsigned integer.

On `char*`, `char**`, and binary search:

As we've seen in lectures 7-10, the type `char*` is a pointer to a null-terminated array of `char`s, which we also call a "string". The naive implementation of our dictionary is simply a `char**`: an array of `char*`'s. Each pointer in the array points to a string (`char*`) containing a single word. And our `check_spelling` operation is then a **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.

Extra Info on Types

More info on how types work in the provided file `Utils.c` can be found in the Extras section, which is not required to finish the exam.

Technical Requirements

Compile-time errors

To start this exam, simply `cd` into 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 in `Spellchecker.c` and `count_typos.c`. You'll want to include some mix of the following, which means you'll need to figure out which ones to include. Here's what we suggest doing for the various headers files you can include:

- C standard library header files: refer to man pages, online documentation, or the error message from `gcc`
- Provided header files: read their doc comments and/or their code to see how each of the code files are connected.

It should be a relatively simple process to get the code to compile.

Runtime errors

Most of the work for this exam will happen here.

The provided code is designed to be **infested** with all kinds of memory errors that programmers can easily make due to human error. 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 exam, we have a few tips for getting started:

- `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, i.e., try running `valgrind` without them, and only add each option incrementally if `valgrind` doesn't provide enough information to debug.
- 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.

Here are some examples of the command formats:

- Running `count_typos` *without* an output file:
  `./count_typos dict/tiny.txt books/simple.txt`
- Running `count_typos` *with* an output file:
  `./count_typos dict/tiny.txt books/simple.txt output.txt`

- Running `valgrind` on `count_typos`:
  `valgrind --leak-check=full ./count_typos dict/tiny.txt books/simple.txt`

Goals

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

- The code needs to compile *without* any warnings by running `make`.
- There should be *no* memory errors.
- Your program's behavior needs to **exactly match** the behavior of the `solution_binary` given to you in the starter code. You can compare them by running `./compare_to_solution.sh`. If any permission error occurs, please run `chmod +x compare_to_solution.sh`.
- **Remember to run `exit` after you finish**. This will trigger the script logging like in the former exercises and let us be able to read your inputs for possible partial credits.

Gdb commands and tips

Command	Description
`break <func>`	this sets a breakpoint at the beginning of func()
`continue`	continue execution past the breakpoint
`kill`	stop the execution of the program while paused at a breakpoint
`clear <breakpoint>`	remove a breakpoint
`next`	execute the next line of your program
`step`	execute and step into a called function

Looking at Variables

Let's say you're looking at the function helloWorld(), which takes an integer argument a, and have set a breakpoint at the beginning of the function. You can use the print command to examine the value of the parameter passed to this call of the function:

```
(gdb) print a
```

You can also use the print function to examine the value of any variable in your current scope.

You can print the current address of a variable using the print command and &. This is particularly helpful for assignments that use pointers such as SymTable.

```
(gdb) print &i – prints address of variable i`
```

Looking at the Stack

You can display a call stack trace using the where command. This is particularly helpful for debugging a segfault.

```
(gdb) where
```

Quitting

You can quit gdb using the following commands:

```
(gdb) quit
```

*Extras

All of the content in this section is not strictly required, but may be helpful.

More on Types in C

You only need to read the following if you want to understand `Utils.c`.

On `int`, `char`:

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.

We can explore this through the following series of examples. 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, using `int`s:

```
int i = '0';
```

is identical to this one:

```
int i = 48;
```

And so are these two, using `char`s:

```
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 the
character '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` (End of file); it just so happens that `EOF` is of type `int`.

Alternatively, we can use `feof()` to detect `EOF` as well.