CSCI 4210 — Operating Systems
Homework 1 (document version 1.0)
Files, Strings, and Memory Allocation in C

- This homework is due by 11:59PM ET on Wednesday, February 10, 2021
- This homework is to be done individually, so **do not share your code with anyone else**
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v18.04.5 LTS and `gcc` version 7.5.0 (`Ubuntu 7.5.0-3ubuntu1~18.04`)

Hints and reminders

To succeed in this course, pay close attention to the lower-level details of C. This includes allocating exactly the number of bytes you need regardless of whether you use static or dynamic memory allocation. This also includes deallocating dynamically allocated memory (via `free()`) at the earliest possible point in your code, as well as closing any open file descriptors as soon as you are done with them.

Another key to success is to always read and re-read the corresponding `man` pages for library functions, system calls, etc. To better understand how `man` pages are organized, check out the `man` page for `man` itself!

Homework specifications

In this first homework, you will use C to implement a rudimentary cache of words, which will be populated with words read from a series of input files. Your cache must be a dynamically allocated hash table of a fixed size that handles collisions by simply replacing the existing word with the new word.

The goal of this assignment is to increase your fluency in C on Linux, in particular handling strings, working with pointers and pointer arithmetic, and dynamically allocating (and re-allocating) memory.

To emphasize and master the use of pointers, **you are not allowed to use square brackets** anywhere in your code! If a `'['` or `']'` character is detected, including within comments, you will receive a zero for this assignment. (Ouch!)

To avoid extraneous square brackets, use the command-line `grep` tool as shown below. Can you combine this into one `grep` call? As a hint, check out the `man` page for `grep`.

```
bash$ grep '\[' *.c
...
bash$ grep '\]' *.c
...
```

Command-line arguments and memory allocation

The first command-line argument specifies the size of the cache, which therefore indicates the size of the dynamically allocated array that you must create. Use `calloc()` to create an array of character pointers. And use `atoi()` (or `strtol()`) for the conversion from a string to an integer.

Next, your program must open and read the regular files specified by the remaining command-line arguments (in left-to-right order). Your program must parse all words (if any) from each given file, determine the cache array index for each word (in the order each word is discovered), then store the word in the cache. If a collision occurs, simply replace the pre-existing word.

To read in each word from each given file, use exactly one dynamically allocated character array of size `128`. You can therefore assume that each word is no more than `127` bytes long (since you want to save a byte to store the end-of-string `'\0'` character).

Initially, your cache is empty, meaning it is an array of `NULL` pointers, since `calloc()` will zero out the allocated memory for you. Storing each word therefore also requires dynamic memory allocation. For this, use `calloc()` if the cache array slot is empty; otherwise, to replace an existing word, use `realloc()` followed by `strcpy()`. Even if the size of the old word is the same as the size of the new word, always call `realloc()`.

Be sure to calculate the number of bytes to allocate as the length of the given word plus one, since strings in C are implemented as `char` arrays that end with a `'\0'` character.

You are **not** allowed to use `malloc()` anywhere in your code. And be sure to use `free()` to deallocate all dynamically allocated memory. Memory leaks are unacceptable.

What is a word and how do you "hash" it?

For this assignment, words are defined as containing only alpha characters and consisting of at least three characters in length. All other characters serve as word delimiters. Further, words are case sensitive (e.g., `Lion` is different than `lion`).

To determine the cache array index for a given word (i.e., to properly "hash" the word), write a separate function called `hash()` that calculates the sum of each ASCII character in the given word as an `int` variable, then applies the "mod" operator to determine the remainder after dividing by the cache array size.

As an example, the valid word `Meme` consists of four ASCII characters, which sum to $77 + 101 + 109 + 101 = 388$. If the cache array size was 17, for example, then the array index for `Meme` would be the remainder of 388/17 or 14.

Required Output

When you execute your program, you must display a line of output for each word that you encounter in each file. And for each word, display the cache array index and whether you called `calloc()` or `realloc()`.

As an example, below is sample output that shows the format you must follow:

```
Word "Once" ==> 15 (calloc)
Word "when" ==> 9 (calloc)
Word "Lion" ==> 11 (calloc)
Word "was" ==> 8 (calloc)
Word "asleep" ==> 5 (calloc)
Word "little" ==> 8 (realloc)
Word "Mouse" ==> 11 (realloc)
Word "began" ==> 16 (calloc)
Word "running" ==> 4 (calloc)
...
```

Further, when you have finished processing all input files, show the contents of the cache by displaying a line of output for each non-empty entry in the cache. Use the following format:

```
Cache index 0 ==> "they"
Cache index 1 ==> "gnawed"
Cache index 2 ==> "King"
Cache index 3 ==> "LITTLE"
Cache index 4 ==> "went"
Cache index 5 ==> "PROVE"
Cache index 6 ==> "sad"
Cache index 7 ==> "tree"
Cache index 8 ==> "little"
Cache index 9 ==> "said"
Cache index 10 ==> "MAY"
Cache index 11 ==> "Mouse"
Cache index 12 ==> "him"
Cache index 13 ==> "FRIENDS"
Cache index 14 ==> "GREAT"
Cache index 15 ==> "the"
Cache index 16 ==> "began"
```

Error handling

If improper command-line arguments are given, report an error message to `stderr` and abort further program execution. In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then aborting further program execution. Only use `perror()` if the given library or system call sets the global `errno` variable.

Error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty.

Note that this assignment will be available on Submitty a minimum of three days before the due date. Please do not ask when Submitty will be available, as you should first perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, make use of the `DEBUG_MODE` technique to make sure that Submitty does not execute any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of q is %d\n", q );
    printf( "here12\n" );
    printf( "why is my program crashing here?!\n" );
    printf( "aaaaaaaaaaaaaggggggggghhhh!\n" );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE hw1.c
```

Second, output to standard output (`stdout`) is buffered. To disable buffered output for grading on Submitty, use `setvbuf()` as follows:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure good results on Submitty, this is a good technique to use.