CSCI 4210 — Operating Systems
Homework 1 (document version 1.1)
Dynamic Memory Allocation, Pointer Arithmetic, and Files

- This homework is due in Submitty by 11:59PM EST on Thursday, January 27, 2022
- You can use at most three late days on this assignment
- This homework is to be done individually, so **do not share your code with anyone else**
- You **must** use C for this assignment, and all submitted 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
- All submitted code **must** successfully compile and run on Submitty, which uses Ubuntu v20.04.3 LTS and `gcc` version 9.3.0 (`Ubuntu 9.3.0-17ubuntu1~20.04`)

Hints and reminders

To succeed in this course, do **not** rely on program output to show whether your code is correct. Consistently allocate exactly the number of bytes you need regardless of whether you use static or dynamic memory allocation. Further, deallocate dynamically allocated memory via `free()` at the earliest possible point in your code. And as covered in initial class videos, make use of `valgrind` to check for errors with dynamic memory allocation and use. Also close any open file descriptors or `FILE` pointers as soon as you are done using them.

Another key to success in this course is to always read (and re-read!) the corresponding `man` pages for library functions, system calls, etc.

Homework specifications

In this first homework, you will use C to implement a rudimentary cache of data, which will be populated with integer values read from a series of input files. Your cache must be a dynamically allocated hash table of a given fixed size that handles collisions by appending each new integer entry to the end of the corresponding integer array. Note that duplicate values must not be added.

To clarify, this hash table is really just a one-dimensional array of `int*` pointers. These pointers should all initially be `NULL`, then set to point to dynamically allocated arrays of integers. Each integer array must be allocated to exactly fit the number of non-duplicate entries.

Since arrays do not have an inherent length in C, use a separate parallel `int` array to maintain the length of each array in your hash table. Since this array will be parallel to your cache, this array must also be dynamically allocated.

No square brackets allowed!

To emphasize and master the use of pointers and pointer arithmetic, **you are not allowed to use square brackets** anywhere in your code! As with our first lab, if a `'['` or `']'` character is detected, including within comments, that line of code will be removed before running `gcc`. (Ouch!)

To detect square brackets, consider using the command-line `grep` tool as shown below.

```
bash$ grep '\[' hw1.c
...
bash$ grep '\]' hw1.c
...
```

Can you combine this into one `grep` call? As a hint, check out the `man` page for `grep`.

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 `int*` array that you must create. Use `calloc()` to create this array of "placeholder" pointers. Also use `calloc()` to create your parallel array of list lengths. And use `atoi()` (or `strtol()`) to convert from a string to an integer on the command line.

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 integers (if any) from each given file. To accomplish this, use `fscanf()`. For each integer read in from the file, determine the cache array index and store the value in the cache. If a collision occurs, append the new value to the end of the existing array of stored values.

Initially, your cache is empty, meaning it is an array of `NULL` pointers (remember that `calloc()` will zero out the allocated memory for you). Storing each value therefore also requires dynamic memory allocation. For this, use `calloc()` if the cache array slot is empty; otherwise, to replace an existing value, use `realloc()` to extend the size of the integer array. Remember you will need to keep track of the length of each array in your hash table via a parallel `int` array.

Note that you are **not** allowed to use `malloc()` anywhere in your code!

To determine the cache array index, use the "mod" operator. As an example, if integer `"1248"` is encountered and the cache array size is 11, the array index for `int` value 1248 would be the remainder of 1248/11 or 5.

(v1.1) Note that only non-negative values should be extracted from the input stream. For example, given substring `"abc123xyz-456"` as part of an input file, extract values 123 and 456. Further, you may assume that given integers will never be larger than 2,147,483,647.

Required Output

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

Given the `numbers.txt` example file shown below, you could run your code with a cache size of 11 as follows:

```
bash$ cat numbers.txt
Hi.  The 10 numbers I'm thinking of are 1, 2, 4, 8, 16, 32, 64, 128,
and then 1248 and lucky 777.  Or unlucky 777.  Thanks.
bash$ ./a.out 11 numbers.txt
```

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.

Below is sample output from the above program execution that illustrates the format you must follow:

```
Read 10 ==> cache index 10 (calloc)
Read 1 ==> cache index 1 (calloc)
Read 2 ==> cache index 2 (calloc)
Read 4 ==> cache index 4 (calloc)
Read 8 ==> cache index 8 (calloc)
Read 16 ==> cache index 5 (calloc)
Read 32 ==> cache index 10 (realloc)
Read 64 ==> cache index 9 (calloc)
Read 128 ==> cache index 7 (calloc)
Read 1248 ==> cache index 5 (realloc)
Read 777 ==> cache index 7 (realloc)
Read 777 ==> cache index 7 (skipped)
========================================
Cache index 1 ==> 1
Cache index 2 ==> 2
Cache index 4 ==> 4
Cache index 5 ==> 16, 1248
Cache index 7 ==> 128, 777
Cache index 8 ==> 8
Cache index 9 ==> 64
Cache index 10 ==> 10, 32
```

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( "aaaaaaaaaaaaagggggggghhhh!\n" );
#endif
```

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

```
bash$ gcc -Wall -Werror -g -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.