ECS 34: Programming Assignment #4

Instructor: Aaron Kaloti

Fall 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Stated what `printLibrary()` and `freeLibrary()` from part #1 are supposed to return. Fixed a typo in part #1 (I put a strikethrough on it so it's obvious). Explicitly stated that you are not provided any files for part #2.
- v.3:
  - In part #2, confirmed that the autograder will never provide unsupported flags to your program.
  - In part #2, noted that the message "No file provided." (shown in the example output) is not printed by `perror()`.
- v.4:
  - `cirQueueCreate()` should return `NULL` if the given capacity is not positive.
  - `cirQueueDequeue()` should return 0 if the queue is empty.
  - For `loadlibrary()` in part #1, you may assume that the file always exists.
- v.5: Autograder details. Gave a more specific point breakdown for part #1. Clarified return value of `countCommonTitles()`.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Tuesday, November 17. Gradescope will say 12:30 AM on Wednesday, November 18, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

You should use the `-Wall` and `-Werror` flags when compiling. The autograder will use these flags when it compiles your program.

3 Grading Breakdown

This assignment will be out of 80 points. Part #2 is worth 30 points, whereas each of part #1 and part #3 is worth 25 points.

Of the 25 points for part #1, 15 is for `loadLibrary()` and `printLibrary()`, 5 is for `countCommonTitles()`, and 5 is for `freeLibrary()`. Needless to say, if you cannot get `loadLibrary()` working, then you cannot get any points for this part.

4 Submitting on Gradescope

You should only submit your C files with the names indicated below. You may be penalized for submitting additional files. You have infinite submissions until the deadline.

During the 10/02 lecture, I talked about how to change the active submission, just in case that is something that you find yourself wanting to do.

You will submit the following C files:

- `library.c`
- `wc.c`
- `circular_queue.c`

4.1 Regarding Autograder

Your output must match mine *exactly*.

There is a description about how to interpret some of the autograder error messages in the directions for the previous two programming assignments. I will not repeat that description here.

The autograder takes longer to run than it did in the previous assignments.

4.1.1 Memory Leaks

Some of the test cases automatically fail if your program has a memory leak. These cases will tell you that there is a leak, if this is the case. Note that failing to close a file also causes a memory leak, perhaps because the `FILE` struct has dynamically allocated memory. Memory leaks will be checked with `valgrind`.

4.1.2 Number of Visible vs. Hidden Cases

Below is the breakdown of visible vs. hidden test cases on each part.

Part	No. Cases	No. Visible	No. Hidden
1	8	4	4
2	15	9	6
3	5	4	1

4.1.3 Test Cases' Inputs

See `part1_visible.c`, `part2_cases.sh`, and `part3_visible.c` on Canvas.

`part1_visible.c` and `part3_visible.c` are compiled in a manner similar to the way in which the C files shown in the examples for parts #1 and #3 were compiled.

`part2_cases.sh` is a shell script. The plan is to go over shell scripts during lecture, but we won't do that until around a week after the due date of this assignment. If you're curious, you can read about shell scripts towards the end of your Linux textbook. That said, you don't need to fully understand shell scripts to understand what's going on in `part2_cases.sh`; you can read through it and determine how `./wc` was run for whichever case number(s) you're curious about.

5 Programming Problems

5.1 Restrictions

Since you will only submit the source files to the autograder, **you are not allowed to modify the header files (`library.h` and `circular_queue.h`).**

Below is a list of headers that you *are* allowed to include. You may not need all of these.

- `library.h`
- `circular_queue.h`
- `<stdio.h>`
- `<stdlib.h>`
- `<string.h>`
- `<limits.h>`
- `<stdbool.h>`
- `<ctype.h>`
- `<assert.h>`

The autograder will penalize you if your code has memory leaks or fails to close any file that it opens.

5.2 Part #1: Library

Before starting this part, you should quickly look at the contents of `library.h`. Here, we see the definitions of two structs. First, there is `struct Book`, which represents – you guessed it – a book. Then, there is `struct Library`, which consists of nothing more than an array of instances of `struct Book` and the length (`numBooks`) of this array. In a file called `library.c`, you will implement the four functions that are declared in `library.h`.

The first function, `loadLibrary()`, creates and returns (a pointer to) an instance of `struct Library`, based on the contents of the named file. The named library data file will always be a CSV file (and will always exist). In a CSV file, each line is called a "record", and each line consists of comma-separated values. In the case of the library data file, each record/line corresponds to a book, and each line contains the title, author(s), and year of the book, all separated by commas. If there are multiple authors, then their names won't be separated by commas, which means you shouldn't have to worry/think about multiple authors at all. In what could be considered a violation of the CSV format, each library data file will always start with a line that contains the number of books/records in the file. You may assume that there is always at least one book and that the file is always properly formatted. This function should use the data file to properly initialize the members (including the array) of the new instance of `struct Library`.

You may assume that the same title will never appear twice in a given ~~title~~ file. (This is an assumption that matters when implementing `countCommonTitles()`.)

See the examples below for how `printLibrary()` behaves.

`countCommonTitles()` takes two instances of `struct Library` and returns the number of book titles that the two libraries have in common. The function should return -1 if given a null pointer, but the autograder will not check this.

`freeLibrary()` takes a pointer to an instance of `struct Library` and deallocates all dynamically allocated memory associated with this instance.

Each of `printLibrary()` and `freeLibrary()` should return 0 if given a null pointer. Otherwise, they should return 1.

In `library.h`, there are two macros `MAX_LINE_LEN` and `BUF_LEN`. No line in the file whose name is given to `loadLibrary()` will ever have more than 300 visible characters, and 300 is the value of `MAX_LINE_LEN`. If you choose to use `fgets()`, you may find `BUF_LEN` helpful.

Below are examples of how your code should behave.

```
$ cat try_library.c
#include "library.h"

#include <stdio.h>

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        fprintf(stderr, "Did not provide two library files.\n");
        return 1;
    }
    char* filename1 = argv[1];
    char* filename2 = argv[2];
    struct Library* lib1 = loadLibrary(filename1);
    struct Library* lib2 = loadLibrary(filename2);
    printf("=== Library 1 ===\n");
    printLibrary(lib1);
    printf("=== Library 2 ===\n");
    printLibrary(lib2);
    printf("\nNo. common titles: %d\n", countCommonTitles(lib1, lib2));
    freeLibrary(lib1);
```

```
    freeLibrary(lib2);
}
$ gcc -Wall -Werror library.c try_library.c -o try_library
$ ./try_library
Did not provide two library files.
$ cat library_files/lib1.csv
4
The Linux Command Line,William Shotts,2019
C++ Crash Course,Josh Lospinoso,2020
Compilers,Aho et al.,2007
Secure Coding in C and C++,Robert C. Seacord,2013
$ cat library_files/lib2.csv
6
Understanding Cryptography,Christof Paar and Jan Pelzl,2010
Automating the Boring Stuff with Python,Al Sweigart,2020
Secure Coding in C and C++,Robert C. Seacord,2013
The Linux Command Line,William Shotts,2019
Managing Projects with GNU Make,Robert Mecklenburg,2005
Programming with 64-Bit ARM Assembly Language,Stephen Smith,2020
$ ./try_library library_files/lib1.csv library_files/lib2.csv
=== Library 1 ===
The Linux Command Line by William Shotts (2019)
C++ Crash Course by Josh Lospinoso (2020)
Compilers by Aho et al. (2007)
Secure Coding in C and C++ by Robert C. Seacord (2013)
=== Library 2 ===
Understanding Cryptography by Christof Paar and Jan Pelzl (2010)
Automating the Boring Stuff with Python by Al Sweigart (2020)
Secure Coding in C and C++ by Robert C. Seacord (2013)
The Linux Command Line by William Shotts (2019)
Managing Projects with GNU Make by Robert Mecklenburg (2005)
Programming with 64-Bit ARM Assembly Language by Stephen Smith (2020)

No. common titles: 2
$ valgrind ./try_library library_files/lib1.csv library_files/lib2.csv
==14690== Memcheck, a memory error detector
==14690== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14690== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==14690== Command: ./try_library library_files/lib1.csv library_files/lib2.csv
==14690==
=== Library 1 ===
The Linux Command Line by William Shotts (2019)
C++ Crash Course by Josh Lospinoso (2020)
Compilers by Aho et al. (2007)
Secure Coding in C and C++ by Robert C. Seacord (2013)
=== Library 2 ===
Understanding Cryptography by Christof Paar and Jan Pelzl (2010)
Automating the Boring Stuff with Python by Al Sweigart (2020)
Secure Coding in C and C++ by Robert C. Seacord (2013)
The Linux Command Line by William Shotts (2019)
Managing Projects with GNU Make by Robert Mecklenburg (2005)
Programming with 64-Bit ARM Assembly Language by Stephen Smith (2020)

No. common titles: 2
==14690==
==14690== HEAP SUMMARY:
==14690==     in use at exit: 0 bytes in 0 blocks
==14690==   total heap usage: 29 allocs, 29 frees, 11,029 bytes allocated
==14690==
==14690== All heap blocks were freed -- no leaks are possible
==14690==
==14690== For counts of detected and suppressed errors, rerun with: -v
==14690== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

5.3 Part #2: `wc`

You are not provided any files for this part. You have to create `wc.c` from scratch.

In a file called `wc.c`, you will implement a program that is very similar to the built-in `wc` Linux command. Recall that this command prints the number of characters, number of words, and number of lines in the given file or files. If more than one file is provided, then a total (of the number of characters/words/lines across all files) is also printed. If a file cannot be

opened, then you should use `perror()` to print the appropriate error message to standard error.

Flags: If you do not want to print the default information (number of characters/words/lines), then you can provide one or more flags in order to override the default behavior. The built-in `wc` Linux command supports several flags that you can examine by doing `man pages`. In our version, we will only support the following flags (short version and long version for each):

- `-m` (`--chars`): number of characters[1].
- `-w` (`--words`): number of words. A word is defined as consecutive non-whitespace characters.
- `-l` (`--lines`): number of lines.

The files may not be specified after the flags. They could be mixed up in their order. A few of the examples below show this.

Your program should also support redundant specification of flags (e.g. specifying both `-l` and `--lines`, or specifying `-w` three times, or doing something like `-mwmmw`.).

Your program should support both ways of specifying multiple short versions of flags (e.g. both `-mw` and `-m -w` should be supported).

Update: The autograder will never provide unsupported flags to your program, e.g. `-x`, `--fake`.

I personally found `fgetc()` and `isspace()` useful in this part, but you may prefer a different way.

Below is what the program (meaning the `main()` function itself) should return:

- 0: no issues.
- 1: at least one named file could not be opened.
- 2: no file was provided.

You may assume that the provided flags are always properly formed, e.g. no ~~meaningless~~ unsupported flag will be used.

When printing out the number of whatever is desired, don't worry about maintaining the semi-tabular format that the real `wc` does. Pay close attention to how my verison (`./wc`) prints in the examples below; that is how yours should print.

Below are examples of how your code should behave. To give you a comparison between the built-in `wc` Linux command and the one that you will implement, I sometimes show the output of each. Doing `wc` runs the built-in command, whereas doing `./wc` runs the executable that we compiled and that is in the current directory. `echo $?` prints the return value of the last command/program that was run on the command line.

```
$ cat foo.txt
Hi there
How are you
$ cat bar.txt
What is up?

Blah
blah blah
blah
$ gcc -Wall -Werror wc.c -o wc
$ ./wc                              # This message is not / cannot be printed by perror().
No file provided.
$ ./wc 2> /dev/null
$ echo $?
2
$ wc foo.txt
 2  5 21 foo.txt
$ ./wc foo.txt
foo.txt: 2 5 21
$ wc foo.txt bar.txt
 2  5 21 foo.txt
 5  7 33 bar.txt
 7 12 54 total
$ ./wc foo.txt bar.txt
foo.txt: 2 5 21
bar.txt: 5 7 33
total: 7 12 54
$ ./wc -m foo.txt
foo.txt: 21
$ ./wc -l foo.txt
foo.txt: 2
$ ./wc -l -m foo.txt
foo.txt: 2 21
$ ./wc -lm foo.txt
foo.txt: 2 21
```

[1] I agree that it is counterintuitive that the short version is not `-c`, but this is true for the real `wc`.

```
$ ./wc -lm --words foo.txt
foo.txt: 2 5 21
$ ./wc --lines --words foo.txt
foo.txt: 2 5
$ ./wc -mm -mwm foo.txt
foo.txt: 5 21
$ ./wc --lines foo.txt bar.txt
foo.txt: 2
bar.txt: 5
total: 7
$ ./wc --lines -w foo.txt bar.txt
foo.txt: 2 5
bar.txt: 5 7
total: 7 12
$ echo $?
0
$ ./wc foo.txt nonexistent1
foo.txt: 2 5 21
fopen: No such file or directory
total: 2 5 21
$ ./wc foo.txt nonexistent1 bar.txt nonexistent2
foo.txt: 2 5 21
fopen: No such file or directory
bar.txt: 5 7 33
fopen: No such file or directory
total: 7 12 54
$ ls -l restricted.txt
--w------- 1 aaronistheman aaronistheman 0 Nov 10 06:32 restricted.txt
$ ./wc restricted.txt
fopen: Permission denied
$ ./wc foo.txt restricted.txt
foo.txt: 2 5 21
fopen: Permission denied
total: 2 5 21
$ ./wc foo.txt --chars bar.txt -l
foo.txt: 2 21
bar.txt: 5 33
total: 7 54
$ ./wc foo.txt -l -m nonexistent -l bar.txt
foo.txt: 2 21
fopen: No such file or directory
bar.txt: 5 33
total: 7 54
$ echo $?
1
$ ./wc -l --chars
No file provided.
$ ./wc -l --chars 2> /dev/null
$
```

5.4 Part #3: Circular Queue

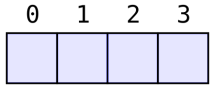
You should review queues (i.e. FIFO queues, not priority queues) from ECS 32B.

When dealing with a queue, it would be nice if enqueueing and dequeueing both took constant ($O(1)$) time. It's easy to implement a queue that sacrifices one of these (e.g. enqueueing takes linear ($\Theta(n)$) time while dequeueing still takes constant time), and you may have seen implementations like this in ECS 32B. In this part of the assignment, you will implement a queue that supports three operations – enqueueing (`cirQueueEnqueue()`), dequeueing (`cirQueueDequeue()`), and finding the number of elements (`cirQueueLength()`) – *all in constant time*. You will be heavily penalized, if not outright given a zero on this part, if you violate the constant time constraint.

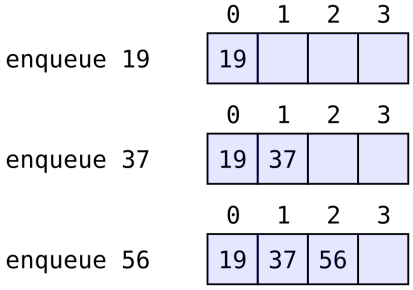
You may not use a linked list to implement your queue. We will explore implementing linked lists in C most likely in the next programming assignment.

If we cannot use a linked list as the underlying implementation of the queue, then that means we must use an array. You are going to implement what can be called a "circular queue" for reasons that will be explained. The underlying array will have a size that will never change (we'll call it the *capacity*), regardless of what sequence of enqueue/dequeue operations occurs[2]. As an example, here is an empty circular queue with capacity 4.

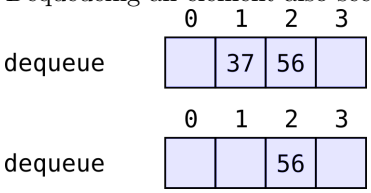
[2]If the size did change, then we would not be able to argue that both operations take constant time. For those familiar with amortized analysis, we could make an argument that both operations take amortized constant time, but that's not what we want here; we want *actual* constant time, not amortized.



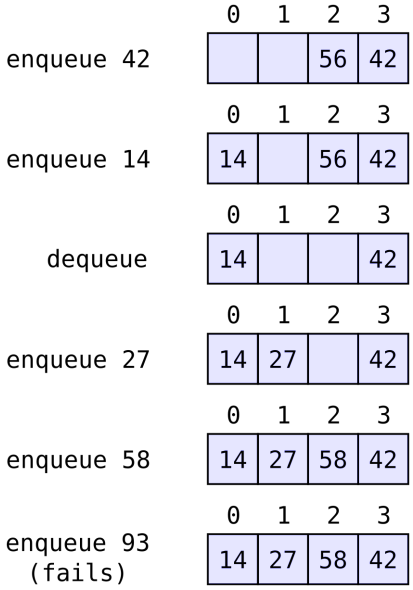
If we enqueue 19, then 37, and then 56, we get the below. Each of these operations seems to easily take constant time, since all that we are doing is filling in an array slot.



Dequeueing an element also seems to easily take constant time. We can dequeue twice to get the below.



If an enqueue operation would result in inserting the element past the end of the array, we instead insert the element at the beginning of the array, as demonstrated below.



As demonstrated at the very end, if we attempt to enqueue into a full circular queue, the attempt should fail.

In other words, we can exploit our commitment to a fixed capacity for the underlying array in order to achieve constant time operations.

If you look at `circular_queue.h`, you will see the `typdef`'d declaration of `struct CirQueue`, as well as the declarations of many supporting functions. In the provided skeleton version of `circular_queue.c`, you can see the definition of `struct CirQueue`. As will be explained during the 11/13 lecture, when the definition is in the source file instead of the header file, the members of the `struct` are unknown outside of `circular_queue.c`. Any outside interaction with an instance of `struct CircularQueue` must be done through the supporting functions, as is shown in the examples below. You get to decide the member variables of `CirQueue`.

Below is a description of each function that you will implement. As a reminder, empty definitions of these functions are already provided in the `circular_queue.c` file provided on Canvas, and you can modify `circular_queue.c` but *not* `circular_queue.h`.

- `CirQueue* cirQueueCreate(int capacity)`: Creates and returns an instance of `CirQueue`, with the underlying array initialized based on the argument provided. Returns `NULL` if the given capacity was not positive.
- `int cirQueueDestroy(CirQueue* queue)`: Deallocates any dynamically allocated memory associated with the given pointer. Returns 0 if given a null pointer. Returns 1 if success.

- `int cirQueueEnqueue(CirQueue* queue, int val)`: If given a null pointer or if the queue is already full, returns 0 immediately. Otherwise, inserts the second argument into the appropriate spot and returns 1.
- `int cirQueueDequeue(CirQueue* queue, int* val)`: Removes whatever element was least recently inserted and places it into the `int` referenced by `val`. Returns 0 if given a null pointer or if the queue is empty (in which case the `int` referenced by `val` shouldn't be modified). Returns 1 if success. *Hint*: Use lazy deletion.
- `int cirQueueLength(const CirQueue* queue)`: Returns the length of the queue (i.e. the number of elements).

As stated above, `cirQueueEnqueue()`, `cirQueueDequeue()`, **and** `cirQueueLength()` **must all take constant time.** One of the TAs or I will check if your queue implementation obeys the above restrictions and is in fact a circular queue. Below are examples of how your code should behave.

```
$ cat try_circular_queue.c
#include "circular_queue.h"

#include <stdio.h>

int main()
{
    CirQueue* queue = cirQueueCreate(4);
    int val = 0;
    printf("1. Length: %d\n", cirQueueLength(queue));
    cirQueueEnqueue(queue, 19);
    cirQueueEnqueue(queue, 37);
    cirQueueEnqueue(queue, 56);
    printf("2. Length: %d\n", cirQueueLength(queue));
    cirQueueDequeue(queue, &val);
    printf("Dequeue: %d\n", val);
    cirQueueDequeue(queue, &val);
    printf("Dequeue: %d\n", val);
    printf("3. Length: %d\n", cirQueueLength(queue));
    cirQueueEnqueue(queue, 42);
    cirQueueEnqueue(queue, 14);
    cirQueueDequeue(queue, &val);
    printf("Dequeue: %d\n", val);
    cirQueueEnqueue(queue, 27);
    cirQueueEnqueue(queue, 58);
    printf("Enqueue failure retval: %d\n",
        cirQueueEnqueue(queue, 93));
    cirQueueDequeue(queue, &val);
    printf("Dequeue: %d\n", val);
    cirQueueDequeue(queue, &val);
    printf("Dequeue: %d\n", val);
    cirQueueDequeue(queue, &val);
    printf("Dequeue: %d\n", val);
    cirQueueDequeue(queue, &val);
    printf("Dequeue: %d\n", val);
    printf("4. Length: %d\n", cirQueueLength(queue));
    cirQueueDestroy(queue);
}
$ gcc -Wall -Werror try_circular_queue.c circular_queue.c -o try_circular_queue
$ ./try_circular_queue
1. Length: 0
2. Length: 3
Dequeue: 19
Dequeue: 37
3. Length: 1
Dequeue: 56
Enqueue failure retval: 0
Dequeue: 42
Dequeue: 14
Dequeue: 27
Dequeue: 58
4. Length: 0
$ valgrind ./try_circular_queue
==16692== Memcheck, a memory error detector
==16692== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16692== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==16692== Command: ./try_circular_queue
==16692==
1. Length: 0
2. Length: 3
Dequeue: 19
```

```
Dequeue: 37
3. Length: 1
Dequeue: 56
Enqueue failure retval: 0
Dequeue: 42
Dequeue: 14
Dequeue: 27
Dequeue: 58
4. Length: 0
==16692==
==16692== HEAP SUMMARY:
==16692==     in use at exit: 0 bytes in 0 blocks
==16692==   total heap usage: 3 allocs, 3 frees, 1,064 bytes allocated
==16692==
==16692== All heap blocks were freed -- no leaks are possible
==16692==
==16692== For counts of detected and suppressed errors, rerun with: -v
==16692== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

UCDAVIS
COMPUTER SCIENCE