**CSCI 4210 — Operating Systems**
**Lab 1 (document version 1.0)**
**Memory Management and I/O in C**

- Given that we are not in-person until the week starting January 24, this lab is due in Submitty by 11:59PM EST on **Thursday, January 20, 2022**
- This lab consists of practice problems and problems to be handed in for a grade; graded problems are to be done individually, so **do not share your work on graded problems with anyone else**
- For all lab problems, take the time to work through the corresponding video lecture(s) to practice, learn, and master the material; while the problems posed here are usually not exceedingly difficult, they are important to understand before attempting to solve the more extensive assignments in this course
- 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`)

## Practice problems

Work through the practice problems below, but do not submit solutions to these problems. Feel free to post questions, comments, and answers in our Discussion Forum.

1. Run the `scanf.c` example from January 11 on your own Linux platform. How many bytes of input does it take for `name` to cause a buffer overflow? How many bytes of input does it take for the buffer overflow to cause a segmentation fault and/or stack smashing error?

   Next, run the `scanf-v2.c` and `fgets.c` examples from January 14. Extend the `fgets.c` example to read in three lines of input, with memory for each line statically allocated to be 32 bytes. The first line is the user's name, the second line is the user's email address, and the third line is the user's home town.

   For each line of input, use `fgets()` to guard against buffer overflow. Also, be sure to "consume" the entire line from the input stream before moving on to the next prompt.

   Finally, uncomment the last prompt that asks the user to enter a number. Be sure that this last bit of functionality works, i.e., that the square root of the given input is displayed.

2. Run the `dynamic-allocation.c` example from January 14. When run with `valgrind`, why are there three dynamic memory allocations when the code only has two `malloc()` calls? How many bytes are dynamically allocated in this third "hidden" call to `malloc()`?

   Further, when you run the `dynamic-allocation.c` example, do you see the same output on your Linux instance? If not, why not?

   Why does the output appear as shown below (from the lecture video), in particular starting on the fifth line of output?

```
sizeof path is 8
path is "/cs/goldsd/s22/" (strlen is 15)
path2 is "AAAAAAAAAAAAAAAA" (strlen is 16)
path is "/cs/goldsd/s22/blah/BLAH/BlAh/blaH/meme" (strlen is 39)
path2 is "aH/meme" (strlen is 7)
munmap_chunk(): invalid pointer
Aborted (core dumped)
```

   Modify the given code by correcting it such that it allocates the minimum number of bytes required for `path` and `path2`. Test this via `valgrind` or `drmemory` to be sure there are no invalid reads or writes.

3. In the code below, how many bytes are dynamically allocated for each call to `malloc()` and `calloc()`?

   For each expression in which we use pointer arithmetic, how many bytes are added to the original pointer?

   What is the exact output of the given code? Can we definitively predict the output or are some values unpredictable?

   Finally, add the appropriate calls to `free()` to ensure no memory leaks.

```
char * name = malloc( 16 );
int * x = calloc( 1, sizeof( int ) );
int * numbers = calloc( 32, sizeof( int ) );
double * values = calloc( 32, sizeof( double ) );

sprintf( name, "ABCD-%04d-EFGH", *x );
printf( "%s\n", name + 3 );
printf( "%d", *(numbers + 5) );
printf( "%lf\n", *(values + 5) );
```

## Graded problems

Complete the problems below and submit via Submitty for a grade. Please do not post any answers to these questions. All work on these problems is to be your own.

1. For this problem, you will place all of your code in a `reverse.h` header file. This file will be included by hidden code on Submitty and compiled as follows:

   ```
   gcc -Wall -Werror lab1-q1.c
   ```

   Review the `reverse()` function shown below to first understand how it works.

   ```
   char * reverse( char * s )
   {
     char buffer[128];
     int i, len = strlen( s );
     for ( i = 0 ; i < len ; i++ ) buffer[i] = s[len-i-1];
     buffer[i] = '\0';
     for ( i = 0 ; i <= len ; i++ ) s[i] = buffer[i];
     return s;
   }
   ```

   Rewrite the `reverse()` function by using dynamic memory allocation for `buffer`. More specifically, do away with the 128-byte buffer and allocate exactly the number of bytes that you need to store the string. Be sure that you check this via `valgrind` or `drmemory`.

   Also, replace all square brackets with pointer notation, i.e., only use pointer arithmetic in your implementation. **Code containing square brackets, including comments, will be automatically deleted on Submitty!**

2. Write a program called `chunk.c` that accepts two command-line arguments `n` and `filename`. Use `open()`, `read()`, `lseek()`, and `close()` to extract `n`-byte chunks from `filename`, skipping every other chunk by using `lseek()`. Read the `man` pages for these functions.

   Display these chunks to `stdout`, delimiting them with a hyphen `'-'` character. Append one newline `'\n'` character at the end of your program execution.

   As an example, given an input file consisting of the English alphabet in lowercase, plus a newline character, your output should produce the results shown below.

```
bash$ cat infile.txt
abcdefghijklmnopqrstuvwxyz
bash$ wc -c infile.txt
27 infile.txt
bash$ ./a.out 7 infile.txt
abcdefg-opqrstu
bash$ ./a.out 4 infile.txt
abcd-ijkl-qrst-yz

bash$
```

   In the example output above, note that the last chunk printed is `"yz\n"`. Read the `man` pages for the `cat` and `wc` programs shown above.

   Just as you did with Question 1 above, replace all square brackets with pointer notation (i.e., use pointer arithmetic). Be sure there are no memory leaks or warnings.

   As a hint, use `atoi()` to convert a string into an integer. Further, use the return value of `read()` to determine when to stop.

## What to submit

Please submit two C source files called `reverse.h` and `chunk.c`. They will be automatically compiled and tested against various test cases, some of which will be hidden.