CSCI 2500 — Computer Organization
Lab 2 (document version 1.0)
Everything is a stream of bytes

- In-person labs start this week with this lab, which is due by the end of your lab section on **Wednesday, February 2, 2022**
- Be sure you attend **only your registered lab section**
- This lab consists of practice problems and problems to be checked off 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

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. In the `buffering.c` code example from the January 27 lecture, if you run this code with `stdout` redirected to an output file, what is the exact contents of the `stdout` buffer when the program crashes?

   Replace all `printf()` calls with calls to `fprintf()` using `stderr` as the first argument, i.e., as the output stream to output to. What is the exact output of this code? If you run this code with `stdout` redirected to an output file, what is the exact contents of the output file when the program crashes?

   Finally, can you figure out how to redirect both `stdout` and `stderr` to the same output file?

2. Rewrite the `file-write.c` code example from the January 27 lecture by asking the user to input integer `n`, which represents the number of values to write to the file. Validate the input, repeatedly asking the user to input a value until valid. Test with large values of `n`; how large can `n` be before your program encounters errors (if ever)?

3. Rewrite the `file-read.c` code example from the January 27 lecture by replacing the `fscanf()` call with calls to `fgets()` and `sscanf()`.

   Next, replace the hard-coded static `arr` array of size 10 with a dynamically allocated array that is sized to hold all values from the given input file. More specifically, use `calloc()` to allocated memory based on the size (in bytes) of the input file. Test with large input files; how large can these files be before your program fails?

Graded problems

Complete the problems below and get each one checked off by one of your lab TAs or mentors. You can ask questions on Submitty, but please do not post answers to these questions. All work on these problems is to be your own.

1. For this problem, place all of your code in `lab2-q1.c` (see below). Compile as follows:

   ```
   gcc -Wall -Werror -Wvla lab2-q1.c -lm
   ```

   The concept of *endianness* plays an important role on your laptop and any computer hardware architecture. Specifically, endianness dictates whether the most significant byte (i.e., "big-end") or least significant byte (i.e., "little-end") is stored as the *first byte* of the value being stored at a given address. Some architectures are big-endian (e.g., Motorola 68K), while other architectures are little-endian (e.g., x86 and its descendants).

   Start by typing in the code below exactly as shown. Step through and understand what's happening in the given code as you type it in, but do not make any changes to it (yet).

   ```
   /* lab2-q1.c */

   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>

   int main()
   {
     /* The 0x prefix below indicates a hexadecimal number */
     int secret = 0x4f414d4c;

     char * cptr = (char *)&secret;
     printf( "%c", *cptr++ );
     printf( "%c", *cptr++ );
     printf( "%c", *cptr++ );
     printf( "%c\n", *cptr++ );

     return EXIT_SUCCESS;
   }
   ```

   How can variable `cptr` point to `secret` if they are different data types? What does `*cptr++` do?

   Next, replace the hexadecimal value of variable `secret` with your own secret four-letter word. To do so, look up each letter in an ASCII table and convert to its corresponding hexadecimal value. An ASCII table is available here: `https://www.asciitable.com/`

   Based on your output, what is the endianness of your system? Verify your answer by running the `lscpu` command in a terminal (and read the corresponding `man` page).

2. For this problem, place all of your code in `lab2-q2.c`. Compile as follows:

   ```
   gcc -Wall -Werror -Wvla lab2-q2.c -lm
   ```

   Based on the endianness of your system, download either the `lab02-data-big.dat` file or the `lab02-data-little.dat` file (each is 744 bytes in size), then attempt to decipher what the file contains. From the shell, try viewing the file in a text editor. Also try using `cat` and `hexdump` to learn what you can about the file. Check the `man` page for `hexdump` to experiment with various flags. Any guesses as to what this file contains?

   Don't read any further until you try to decipher what's in this file!

   Next, determine the size (in bytes) of the data file. To do so, start with the `ls` command. Then, in your code, use the combination of `fseek()` and `ftell()` to determine the size (in bytes) of the file at runtime.

   Assume that the given data file contains an array of `unsigned int` variables. Write a C program to read this data file entirely into heap memory, then display the data using a loop.

   To accomplish this, you must call each of the following functions **exactly once**: `fopen()`, `ftell()`, `calloc()`, `fread()`, `fclose()`, and `free()`. Further, you must call the `fseek()` function **exactly twice**.

   Display the data as follows:

   ```
   DATA POINT #  0: <unsigned-int-value>
   DATA POINT #  1: <unsigned-int-value>
   DATA POINT #  2: <unsigned-int-value>
   ...
   DATA POINT # 99: <unsigned-int-value>
   DATA POINT #100: <unsigned-int-value>
   DATA POINT #101: <unsigned-int-value>
   ...
   ```

   Any guess at this point as to what this file contains?

3. Copy your code for Question 2 above to a new C file called `lab2-q3.c`. Modify this copy by assuming that the given data file contains an array of `unsigned long` variables. Display the data as follows:

   ```
   DATA POINT # 0: <unsigned-long-value>
   DATA POINT # 1: <unsigned-long-value>
   DATA POINT # 2: <unsigned-long-value>
   ...
   DATA POINT # 9: <unsigned-long-value>
   DATA POINT #10: <unsigned-long-value>
   DATA POINT #11: <unsigned-long-value>
   ...
   ```

   Did you figure out what this data file contains yet?

   An important takeaway from this lab is understanding that we can interpret raw data however we like, treating the data as an array of `unsigned int` values, an array of `unsigned long` values, a string of characters, etc. Only one interpretation turns out to be correct, though.