CSE 2421 Lab 2

Learning C, redirection, bit shifting, and bitwise operations

Due: Tuesday 30 May 2023 11:59 PM
Early: 2 bonus points if submitted by Sunday 28 May 2023 11:59 PM
Late: Wednesday 31 May 2023 11:59 PM

We'll be using Carmen's timestamps to determine lateness or not. If you are not enrolled, you may send me an email with the zip file. If you are unable to submit by the deadline, you need to send me a screenshot of your finished files with the clock visible and then *still* submit to Carmen when you can.

This lab contains three parts. Make sure you do all of them!

Something you should do before getting started on the lab is creating your readme file. Simply create a file called "readme" and include your name at the top. As you progress through the lab, you may see different parts (labeled things like **Part 1**). When you see this, write it down in your readme. While in a numbered part of a lab, you'll come across various questions, prepended with a bolded Q and a number (such as **Q5**). When you see this, include the Q and the number, along with you answer to the question(s) in the file. A brief example:

> Name: Rob LaTour
>
> I certify that I completed all of the work myself with no aid from anyone aside from the instructor or the undergraduate graders.
>
> Part 1:
>
> Q1: Don't make fun of me, but my favorite color is gray
>
> Q2: The definitive answer is that Han did shoot first. The original version is the one that captured the heart of the public, and thus the director lost creative control. In this essay I will…

Part 1: Encryption

For this lab, we will be implementing a relatively simple (and limited) encryption/decryption scheme. The first part of the lab requires you to produce a c program that encrypts a plaintext file into another file. I highly recommend breaking this task up into three chunks: first produce an executable that can, through redirection, read an input file character by character and print those characters to an output file. Next, produce an executable that reads in 8 characters from the input file, character by character, then writes those to the output file. Your executable should handle the case where there are fewer than 8 characters in the input file and still work to transfer those characters to the output file. Finally, work on tackling the encryption scheme pictured below.

NOTE: Your code should be written to be as modular as possible. Although the scheme described here reads data in and processes it in chunks of 8 bytes, your program should be (relatively easily) extensible to a scheme that uses larger batches. With this in mind, I recommend creating a header file to define macros rather than using the constant 8 everywhere. You may find it useful to put other things in this header as well!

Our simple encryption scheme is pictured here:

I_1:	0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
I_2:	0	b_1	b_2	b_3	b_4	b_5	b_6	b_7
I_3:	0	c_1	c_2	c_3	c_4	c_5	c_6	c_7
I_4:	0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
I_5:	0	e_1	e_2	e_3	e_4	e_5	e_6	e_7
I_6:	0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
I_7:	0	g_1	g_2	g_3	g_4	g_5	g_6	g_7
I_8:	0	h_1	h_2	h_3	h_4	h_5	h_6	h_7

Figure 1: Before Encryption

I_1:	h_7	a_1	a_2	a_3	a_4	a_5	a_6	a_7
I_2:	b_1	h_6	b_2	b_3	b_4	b_5	b_6	b_7
I_3:	c_1	c_2	h_5	c_3	c_4	c_5	c_6	c_7
I_4:	d_1	d_2	d_3	h_4	d_4	d_5	d_6	d_7
I_5:	e_1	e_2	e_3	e_4	h_3	e_5	e_6	e_7
I_6:	f_1	f_2	f_3	f_4	f_5	h_2	f_6	f_7
I_7:	g_1	g_2	g_3	g_4	g_5	g_6	h_1	g_7

Figure 2: After Encryption

In Figure 1, we have 8 characters (labeled I sub 1 through 8). Each of the lowercase letters in the boxes represents a bit (a 1 or a 0). At a high level, we are simply taking the bits that make up our 8th character and striping those bits through the bits of the preceding characters.

Your task is to create an executable file that will read in 8 characters at a time from an input file, encrypt those following this scheme, and then write the encrypted 7 characters to an output file. That process should be repeated until the end of the file is reached. If the number of characters in the file isn't a multiple of 8 (and it likely won't be), your program should encrypt as many sets of 8 as it can, then at the end it should simply transcribe the remaining

characters without changing them. This task will heavily make use of bit shifting and bitwise operations!

Create a set of test files to make sure your program is producing reasonable output. You can choose exactly what you want your test files to look like, but you'll need to justify why you created them shortly. There are 2 files I will require you to create, however: One is an input file with exactly 8 characters in it. One is the corresponding "expected output" file. Do the encryption scheme by hand and write out this file. When you test your encryption scheme with the 8-character-long input file, you can then use the Linux command diff to ensure that the files are identical. When you're confident that your program is working, run each of your test files through your program and keep them around. They'll be useful in Part 2!

Q1: Briefly justify why you created the test files you're including with your lab. To help you write this answer, consider the following: what potential failure point does this test file cover? What kind of test case does this file represent, a routine, boundary, or challenging case?

Q2: Copy down the 8 characters in your required test file as well as the 7 characters in your "expected output" file. Why did you choose the characters that you did?

Q3: Describe one way that this encryption scheme could be modified (from an algorithmic perspective) quickly and easily in the event that this particular strategy was foiled. Note that there are a lot of possible correct answers to this question!

Q4: As a follow-up to the question above, describe at a high level how we could generalize our code such that an extra input or macro could swap between schemes.

Part 2: Decryption

For the second part of this lab, you're going to create an executable that can decrypt your encrypted text back to plaintext. The procedure will essentially be the inverse of what was done above. The program should read in an input file 7 characters at a time, decrypt those, then repeat until the file ends. Once again, if there are fewer than 7 characters, your program should simply transcribe the characters as written.

If your program is able to read in a full 7 characters, then the stripe of bits labeled by an h in the diagram make up the bits of the 8^{th} character. Consult the diagram above if you are confused! Once again, we will have to do a lot of bit shifting and bitwise operations.

You can use the output of Part 1 to test your implementation of Part 2!

Q1: You may have noticed something unexpected/bad can happen with this scheme as described. Namely, it is possible to correctly follow the encryption scheme, then immediately correctly follow the decryption scheme, and then end up with garbage as a result. How could this happen? Why?

Q2: Describe one way that the problem in Q1 could be fixed. Once again, there are many correct answers here!

Part 3: Making the graders' eyes bleed

This part is largely for fun, but it will also include some miscellaneous questions about the lab as a whole, since it assumes you have finished both Part 1 and Part 2.

Your task is to create an alternative implementation to your code in either Part 1 or Part 2. MAKE SURE THAT YOU CREATE A COPY OF THE FILE BEFORE YOU START EDITING. IT WOULD BE A TRAVESTY FOR YOU TO RUIN SOMETHING YOU ALREADY WORKED ON FOR SO LONG. Your code should be as unreadable as possible while still maintaining the full functionality. For full credit, your code should include at least one goto, break, and continue. Additionally, there should be at least one loop or if construct without a block under it, and at least one conditional that uses an implicit comparison rather than an explicit one. While not an expectation, feel free to rename variables to be have similar and nondescriptive names, delete comments, and put multiple statements on the same line. Please remember, however, that the code must still function identically to whichever of your functions you rewrote. The graders will test for this.

Q1: Did you choose to mangle a copy of your encryption code or your decryption code? Why?

Q2: Answer this question honestly: at the time of writing this question, is it easy for you to read and understand what your mangled code does? Do you think it would be several months from now?

Q3: In this portion of the lab, you took what was (ostensibly) good code and made it hard to read. Do you think that it would be easier or harder to take hard to read code and make it good? Why?

And now for some questions about the lab as a whole:

Q4: Some of you may have noticed that there is an unwritten assumption about the input data that makes a dramatic impact on the functionality of this encryption/decryption scheme. What's the unwritten assumption?

Q5: If this assumption is not true, what part of the process could fail?