Spring 2022 CSE 2421 LAB 2

DUE: Tuesday, February 1st, 2022, 11:30:00 p.m.
10 Bonus Points (i.e.,10% bonus) awarded if correctly submitted by Monday, January 31st at noon.
Any lab submitted up to 24 hours late will be graded and points earned multiplied by 0.75.
Any lab submitted more than exactly 24 hours from the due date will not be graded and scored as a 0.

Objectives/Skills:

- Standard character-based I/O
- Arithmetic/bitwise statements
- while, for, do-while statements
- ASCII character representation
- #ifdef preprocessor statements
- Declaring variables of appropriate types to implement the program
- using the supplied **make** command to create 2 programs from one .c file

REMINDERS:

- This lab is an individual assignment.
- Waiting until the day this lab is due to start working on it would be a very bad idea.
- If you don't read the entire lab description thoroughly, you may miss some important information that will help you.
- Make a point to read the REQUIREMENTS section before writing any code and again, after you have finished the lab. The second time through the REQUIREMENTS section, verify that each of the requirements has been adhered to within your code. You'll get more points that way.

– Every lab requires a README file. For this lab, create your file on stdlinux in a text editor and it should be named LAB2README (***exactly*** this name (case matters), with no "extensions"; Linux can identify the file as a text file without any "extension," such as txt, doc, etc., so do not use these). This file should include the following:

1. Required Header:
   BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE STRICTLY ADHERED TO THE TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY WITH RESPECT TO THIS ASSIGNMENT.

   THIS IS THE README FILE FOR LAB 2.

Student name:

2. Total amount of time in hours (approximate) to complete the entire lab;
3. Short description of any concerns, interesting problems or discoveries encountered, or comments in general about the contents of the lab;
4. Run gdb on your bit_encode1 executable. Enter data here that describes the interim values you calculated as you created your 8-bit key as you read in each separate digit of the 4-digit key. Specify the gdb command you used to print the interim values and how the values were displayed in gdb. For example, what is the value of key after you have read and processed the first digit of the key, then after you have processed the second digit of the key, etc. Printing the key in hexadecimal format might help you determine if the value you are creating is correct.
5. If you run gdb for any reason on bit_encode2, then describe why you chose to do so, what you did, what you saw.

– You should aim to always hand assignments in on time. If you are late (even by less than a minute), you will receive 75% of your earned points for the designated grade as long as the assignment is submitted by 11:30:00 pm the following day (NOTE: This is not necessarily the same as the following class day!), based on the due date listed at the top of this document. If you are more than 24 hours late, you will receive a zero for the assignment and your assignment will not be graded at all (so you will get no feedback on your work). Start early, and work steadily on the lab, and you should be able to hand the assignment in on time without a problem. If you are unable to finish on time, you will need to consider whether it is better to submit on time and get full credit for what you have done, or if it would be better to try to finish and submit one day late with the 25% penalty.

– Any lab submitted that does not build to an executable using the required gcc command (see below) and run without crashing or freezing WILL RECEIVE AN AUTOMATIC GRADE OF ZERO. No exceptions will be made for this rule - to achieve even a single point on a lab your code must minimally compile (without errors or warnings) and execute without crashing or freezing.

– I**t is your responsibility to ensure that your program compiles without errors or warnings and runs on valid input as described without crashing or freezing.**

- We will be taking advantage of a feature of Unix/Linux called **make** to compile executables and to create the .zip file needed to upload to Carmen.

– You are responsible for making sure that your lab submits correctly.

– Make yourself aware of the CODING_STYLE_IN_C file posted on Piazza in the General Resources section.

GRADING CRITERIA

– The code and algorithm are well documented, including an explanatory comment for each function, and comments in the code.

- A comment should be included in the main function of the program including the programmer's name as well as explaining the nature of the problem and an overall method of the solution (what you are doing, not how).

- A comment should be included before each function documenting what the function does (but not details on how it does it).

- A short comment should be included for each logical or syntactic block of statements.

– The program should be appropriate to the assignment, well-structured and easy to understand without complicated and confusing flow of control. We will not usually deduct points for the efficiency of your code, but if you do something in a way which is clearly significantly less efficient, we may deduct some points.

– There is a description in LAB2README of the gdb results asked for both bit_encode1 and bit_encode2.

– The results are correct, verifiable, and well-formatted. The program correctly performs as assigned with both the given input and one other (unknown) input file designed to test boundary conditions within your program.

– **If the grader cannot compile your code using the supplied Makefile with no error or warning messages, you will receive 0 points for the lab. No exceptions.**

REVIEW

Review the lectures with binary values and bitwise operations. Recall that within those bitwise operations, operand 1 and operand 2 could be variables or could be a constant value. For example, if there were declared **unsigned char var1=0**, then you could set bit 6 and bit 1 to **1,** by OR'ing **var1** with 0x42. 0x42==0b 0100 0010. Recall that bit positions start at 0 and increase as you move to the left. 8 bits would have bit positions 0-7, 32 bits would have bit positions 0-31, etc.

Complete Bitwise Homework in Carmen. Working with Q1&2 therein, you should have concluded that we were logically shifting the x value 1 position to the right, but rather than filling the most significant bit with 0, the result had whatever value that was shifted out of x as the new most significant bit. This concept is called "bit rotation". Note these examples: if x=0b 1001 0011, then rotating x one bit to the right results in 0b 1100 1001. if x=0b 0101 1100, then rotating x one bit to the right results in 0b 0010 1110. From the column titles in Q1, you should be able to determine the C language instructions that would allow you to implement this. Q3 asked you to figure out an algorithm to rotate some value x one bit to the left.

LAB DESCRIPTION

PART 1:

If you are not using FastX, use a process like what you did to transfer files between your personal device and stdlinux in lab1 to get a copy of the file **Makefile.Lab2** to your **lab2** directory on stdlinux. You will need to change the name of **Makefile.Lab2** to **Makefile** once you get it to stdlinux.

If you are using FastX, from a window on stdlinux bring up a web browser and navigate to the Piazza page for this class. Go to the Resources->Labs link and find a file called **Makefile.Lab2**. Click on the filename. A window should pop up that says something like:

> You have chosen to open:
> **Makefile.Lab2**
> which is: BIN file (1.0 KB)
> from: https://piazza-resuorces.s3.amazonaws.com
>
> Would you like to save this file?

Click the **Save File** button.

You can now exit out of the web browser.

From your stdlinux window, maneuver to the directory $HOME/cse2421.

1. Create a **lab2** directory.
2. Enter the **lab2** directory.
3. Execute the following command: **cp $HOME/Downloads/Makefile.Lab2 Makefile**
4. Use the **ls** command to verify that you now have a file in your current working directory called Makefile.
5. Use a text editor or the command **cat Makefile** to inspect the contents of the file. Do not make any changes/edits to this file. We will talk about the contents of this file and how it works next week. For the time being, consider it magic.

   This file will allow you to create both executables required for this lab and the .zip file required for this lab. Once you have created a **bit_encode.c** file, this makefile will allow you to create two separate, unique programs **bit_encode1** and **bit_encode2** using only one .c file.

   A. To compile the executable for PART 2, enter the command **make bit_encode1** on the command line.
   B. To compile the executable for PART 3, enter the command **make bit_encode2** on the command line.
   C. To create the .zip file to submit to Carmen, enter the command **make lab2.zip** on the command line.
   D. If you want to do all three of these things at the same time, enter the command **make** on the command line.
   E. If you use the command **make clean**, all executable files and the current .zip file will be deleted from your directory. Your .c files will not be touched.

PART 2 (50%): Mandatory file name: bit_encode.c
Mandatory executable name: bit_encode1

Write a program that implements a bit stream cipher. An elementary level bit stream cipher is an encryption algorithm that encrypts 1 byte of plain text at a time. This cipher uses a given 4-bit bit pattern as the key. The size of the encrypted message that we want to be able to send has a **maximum length of 200 characters**. The 4-bit pattern must be duplicated to an 8-bit value for this to work. (e.g., if we have the 4-bit pattern 0110, then the 8-bit key would be 0110 0110.)

Here is an example:

If we have the ASCII value for the letter 't', then that equates to the hex value 0x74, which represents the binary value 0111 0100. If we bitwise XOR that value with a key, say 0101 0101, then

0111 0100 -> 't'

0101 0101 -> XOR with key } encryption

0010 0001 -> encrypted value 0x21

If we XOR the encrypted value with the key, we get our ASCII value back.

0010 0001 -> encrypted value 0x21

0101 0101 -> XOR with key } decryption

0111 0100 -> 't'

The bit stream encryption cipher that your program must implement will do a little more than this. We are going to add an alternating bit rotation twist to this algorithm. The alternating bit rotation algorithm that must be implement is that once you have the encrypted value (0x21, in the example above), if it is the 1st character of the message we are encrypting, it must be rotated one bit to the left (in our example above that would mean 0b 0010 0001 would become 0b 0100 0010 or 0x42). The encrypted value for the 2nd character of the message would be rotated one bit to the right, 3rd char rotated left, 4th rotated right and so forth. As a further example, if 't' is rotated to the right one position, that would mean 0b 0010 0001 would become 0b 1001 0000 or 0x90.

REQUIREMENTS

In this program:

1. You must adhere to the requirements in the file: **Doing_Labs_in_this_Class**.
2. you must create at least 4 different functions: **main()**, **create_key()**, **rotate_right()**, and **rotate_left()**. You can create other functions to read in the clear text message or to print array values in hexadecimal or some other function(s) that you think would be appropriate if you'd like. The function **create_key()** must read in the 4 digits of the key and return the 8-bit key to the calling function. The functions **rotate_right()** and **rotate_left()** must be passed a single hexadecimal value and return to the calling function that value rotated 1 position right or left. To maintain the 10 line limit, you may need to create additional functions.
3. you must prompt the user to input the clear text to be encrypted. You must use printf() to send the prompt to the user,

4. you must use getchar() to read each ASCII clear text character and store it in a character array. Each character will be a lower-case character.

5. when a ‘\n’ character is detected, this indicates that the last character read in (not the ‘\n’) was the last clear text character of the message to be encrypted,

6. you must use printf() to print out the received clear text so that the user can verify their input. Don’t forget that character strings in C must be null terminated and that means that the array must be big enough to hold the maximum number of characters plus that null terminator.

7. you must use printf() to print out the clear text as hexadecimal numbers rather than ASCII characters. The format should be 10 hexadecimal numbers per row and each number should be represented as 2 (and only 2) hexadecimal digits.

8. You must use printf() to prompt the user for a 4-bit key to encrypt the data (e.g., 0110, 1010, etc.)

9. You must use getchar() to read in each digit of the 4-digit key.

10. You must process each digit of the key immediately. This means that you cannot read in the 4 digits and store them in 4 separate spots in memory then process them as a group.

11. You must create an 8-bit key from those 4 digits using **ONLY** the bitwise OR instruction or **ONLY** the bitwise OR instruction and a shift instruction. For example, if the user specifies 0110, then your internal key must be 0110 0110. Using any other operations to create the key will result in a 50-point reduction in your score for this lab. TO BE VERY CLEAR HERE (SINCE MANY STUDENTS SEEM TO MISUNDERSTAND) FROM THE TIME YOU READ IN THE ASCII VALUE FOR EACH DIGIT OF THE KEY WITH THE getchar() FUNCTION, YOU MAY USE NO OTHER COMPUTATIONS TO CREATE THE 8-BIT KEY EXCEPT WHAT IS SPECIFIED IN THIS REQUIREMENT. NO ADDITION OR SUBTRACTION OR ANY OTHER COMPUTATIONS CAN BE USED.

12. Once you have created an 8-bit key, you must XOR the key with each character of clear text to complete the first step toward creating each character of cipher text.

13. The 2nd step to create the cipher text is to bit-rotate one position left the 1st, 3rd, 5th, etc. characters and bit-rotate one position right the 2nd, 4th, 6th, etc. characters of the message.

14. You must use printf() to print out each hexadecimal cipher text value with 10 values per row. It should be similar to the output below.

```
Sample Output: [jones.5684@cse-fl1 lab2] bit_encode1

enter cleartext: two fat dogs

two fat dogs

hex encoding:

      74 77 6F 20 66 61 74 20 64 6F
```

```
        67 73
enter 4-bit key: 0110
hex ciphertext:
        24 88 12 23 00 83 24 23 04 84
        02 8A
[jones.5684@cse-fl1 lab2]$
```

NOTE!! The input above is not the only input that can be entered in to the program that will be expected to work correctly. It is only an example of input. You can print the hexadecimal numbers as uppercase or lowercase. Your choice.

15. All source code files (.c files) submitted to Carmen as a part of this program must include the following at the top of each file:

```
/*  BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE STRICTLY ADHERED
** TO THE TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY
** WITH RESPECT TO THIS ASSIGNMENT.
*/
```

    If you choose not to put the above comment in your file, you will receive no points for this part of the lab.

16. You must comment your code
17. You must put you name in a comment at the top of the source code file and your Makefile
18. All the code you write must be in the file bit_encode.c.
19. You must correctly use preprocessor directives **#ifdef PROMPT** and **#endif** to complete Part 3.

As long as you adhere to the requirements above, how you choose to write your program is up to you. With each Systems 1 lab you will get more and more freedom with respect to your code. Not using the correct I/O functions as specified above in 3, 4, 6, 7, 8, 9 and 14 will result in a 40-point deduction.

PART 3. (50%). Mandatory file name: bit_encode.c
Mandatory executable name: bit_encode2

After you get bit_encode1 working, introduce **#ifdef PROMPT** and **#endif** preprocessor statements within your bit_encode.c file that modify your program such that it will read clear text and the 4-bit key input without printing any prompts at all and prints no other output other than the cipher text without concern for columns. Input will be coming from a redirected input source on the command line. . It will help you understand how to do this if you inspect the **Makefile** supplied with this lab, look at the comments and observe the options used in the **gcc** statements (such as -D PROMPT). If you previously chose not to read Chapter 14 of the Reek book, you may want to reconsider at this point.

Sample output:

```
[jones.5684@cse-fl1 lab2] bit_encode2 < encode.input

8B 61 87 E2 15 EC BD E5 BD E7 15 EF 89 E1 B9 E7 B1 EC 87 6F A7

[jones.5684@cse-fl1 lab2]$
```

In this example, encode.input contains these two lines:

```
ohio state university
1010
```

LAB SUBMISSION

Always be sure your linux prompt reflects the correct directory or folder where all files to be submitted reside. If you are not in the directory with your files, the following will not work correctly.

You must submit all your lab assignments electronically to Carmen in .zip file format. The .zip file will be created by using the **make** command (See PART 1.) Once you execute the command, you should find a file in your lab2 directory called **lab2.zip**; this is the file that must be uploaded to Carmen.

NOTE:

- It is YOUR responsibility to make sure your code can compile and run on CSE department server **stdlinux.cse.ohio-state.edu,** using required gcc options without generating any errors or warnings or segmentation faults, etc. Any program that generates errors or warnings when compile or does not run without system errors will receive 0 points. No exceptions! This means you may want to copy the verify command from lab1 to your lab2 directory, modify it so that it will make bit_decode1 and bit_decode2 in the test directory.