CSE 2421 LAB 2

Objectives/Skills:

- Standard character based I/O.
- Arithmetic/bitwise statements.
- while, for, do-while statements.
- ASCII character representation.
- #ifdef preprocessor statements.
- using the **make** command.

REMINDERS:

– This lab is an individual assignment.

– Every lab requires a README file. For this lab, 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 PERFORMED ALL OF THE WORK TO CREATE THIS FILE AND/OR DETERMINE THE ANSWERS FOUND WITHIN THIS FILE MYSELF WITH NO ASSISTANCE FROM ANY PERSON (OTHER THAN THE INSTRUCTOR OR GRADERS OF THIS COURSE) AND I HAVE STRICTLY ADHERED TO THE TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY.

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_encode2 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.
5. If you run gdb for any reason on bit_encode1, then describe why you chose to do so, what you did, what you saw.
6. Your opinion of the Makefile tool. [2 sentences]

– 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. No further delays allowed.

– **No errors, No Warnings:** Any lab submitted that does not build to an

executable using the required gcc command (see below) and run without crashing or freezing will be graded with Zero.

– From this lab going forward, we will be taking advantage of a feature of Unix/Linux called makefiles 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 Carmen.

GRADING CRITERIA (approximate percentages listed)

– (10%) 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(s) 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.

– (10%) 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.

– (20%) There is a description in LAB2README of the gdb results asked for both bit_encode1 and bit_encode2.

– (60%) 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 no points.**

LAB DESCRIPTION

PART 1:
From a window on stdlinux bring up a web browser and navigate to the Carmen page for this class and save the **Makefile.Lab2** file or download it.

From your stdlinux window, change 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**
   - *ensure that you copy the file from the right directory to lab2 directory*
4. Use the **ls** command to verify that you now have a file in your current working directory called Makefile.
5. Use a file editor or the command **cat Makefile** to inspect the contents of the file. Do not make any changes/edits to this file.

   This file will allow you to create both executables required for this lab and also the .zip file required for this lab. Once you have created a **bit_encode.c** file,

   A. To compile the executable for PART 2, enter the command **make bit_encode2** on the command line.
   B. To compile the executable for PART 3, enter the command **make bit_encode1** 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_encode2

Write a program that implements an elementary bit stream cipher. An elementary level bit stream cipher is an encryption algorithm that encrypts 1 byte of plain text at a time. This one 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. You must:

1. Prompt the user to input the clear text to be encrypted. You may use printf() to send the prompt to the user,
2. Use the getchar() program to read each ASCII clear text character and store it in a character array. Each character will be a lower case character.
3. When a '\n' character is detected, this indicates that the last character read in, not the '\n', was the last clear text character (i.e., your program should not encode the '\n', it only use it as an indication of ending of the user input).
4. print out the received clear text so that the user can verify their input,
5. 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 hexadecimal digits.
6. Prompt the user for a 4-bit key to encrypt the data (e.g. 0110, 1010, etc.)
7. You have to use getchar() to read in each of the 4 digits of the key.

8. The key will have to be converted within your program from 4-bits to 8-bits. For example, if the user specifies 0110, then your internal key must be 0110 0110.
9. Once you have created an 8-bit key, you must XOR the key with each character of clear text to get each character of cipher text.
10. Print out each hexadecimal cipher text value with 10 values per row. It should be similar to the output of the hex encoding above.

Output might look similar to the following:

```
[abushattal.1@cse-fac1 lab2] bit_encode2
enter cleartext: two fat dogs
Text entered is: two fat dogs
Hex encoding is:
74 77 6F 20 66 61 74 20 64 6F
67 73
enter 4-bit key: 0110
hex ciphertext:
12 11 09 46 00 07 12 46 02 09
01 15
[abushattal.1@cse-fac1 lab2]
```

NOTE!! The input above is not the only input that can be entered into the program that will be expected to work correctly. It is only an example of input.

CONSTRAINTS:

– 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 PERFORMED ALL OF THE ** WORK TO CREATE THIS FILE AND/OR DETERMINE THE ANSWERS FOUND WITHIN THIS
** FILE MYSELF WITH NO ASSISTANCE FROM ANY PERSON (OTHER THAN THE INSTRUCTOR ** OR GRADERS OF THIS COURSE) AND I HAVE STRICTLY ADHERED TO THE TENURES OF THE ** OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY. */
```

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

– You must comment your code

– Be sure your directions to the user are clear so they are sure to enter the input data correctly.

PART 3. (50%). Mandatory file name: bit_encode.c
Mandatory executable name: bit_encode1

After you get bit_encode2 working, introduce #ifdef PROMPT and #endif preprocessor statements into your code (you must use the same file, bit_encode.c)

that change the 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 hex encoding without concern for columns. Input will be coming from a redirected input source from the command line.

Example on the #ifdef #endif:

```
int main()
{
    int x;
    #ifdef PROMPT  /* Checks if the macro is defined then it should execute the
    printf statement, else it will ignore it.*/
        printf("Input Data:\n");
    #endif

    scanf("%d", &x);
    printf("%d",x);
    return 0;
}
```

Sample output for lab 2:

[abushattal.1@cse-fac1 lab2] bit_encode1 < encode.input
12 11 09 46 00 07 12 46 02 09 01 15
[abushattal.1@cse-fac1 lab2]

In this example, encode.input contains these two lines:
two fat dogs
0110

LAB SUBMISSION

Always be sure your Linux prompt reflects the correct directory or folder where all of your 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 can 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.

Your readme file must be submitted to the appropriate Carmen assignment as well as your .zip file being submitted to its Carmen assignment.

NOTE:

- It is your responsibility to make sure your code can compile and run on CSE department server **stdlinux.cse.ohio-state.edu,** using **gcc -std=c99 –pedantic –g** 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!