CSC230 Homework 03

This homework is to be done individually.

You may use **getchar()**, **putchar()**, **printf()**, and **exit()**. All other functions are not allowed for Homework 3.

Learning Outcomes

- **Write small to medium C programs** having several separately-compiled modules.
- Explain what happens to a program during preprocessing, lexical analysis, parsing, code generation, code optimization, linking, and execution, and **identify errors that occur during each phase**. In particular, they will be able to describe the differences in this process between C and Java.
- **Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.**
- **Find and eliminate runtime errors using a combination of logic, language understanding, trace printout,** and gdb or a similar command-line debugger.
- **Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow**.
- **Explain, inspect, and implement programs** using structures such as enumerated types, unions, and constants and **arithmetic, logical, relational, assignment, and bitwise operators**.
- **Trace and reason about variables and their scope in a single function, across multiple functions, and across multiple modules.**
- Allocate and deallocate memory in C programs while avoiding memory leaks and dangling pointers. In particular, they will be able to implement dynamic arrays and singly-linked lists using allocated memory.
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- **Write, debug, and modify programs using library utilities**, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, **standard I/O,** and file I/O.
- **Use simple command-line tools to design, document, debug, and maintain their programs.**
- Use an automatic packaging tool, such as make or ant, to distribute and

maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.
- **Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.**

Program - des.c

In this program, you will be implementing the DES encryption/decryption algorithm using the Electronic Code Book cipher mode. DES stands for *data encryption standard* and is a block cipher algorithm developed by IBM and used for a long time as the standard for document and password encryption and security, though DES is now no longer considered secure. Read more at wikipedia.

C is a useful language to implement DES, since it has the ability to do the bit movements required by the algorithm.

Requirements

The program reads in two command line arguments. The first command line argument is either an **`'e'`** for encrypt or **`'d'`** for decrypt. The second command line argument is an exactly 8-byte key/password. If either command line argument is missing or if the user enters a character other than **`'e'`** or **`'d'`** (the characters **`'E'`** or **`'D'`** would also be considered invalid), a usage description of **`"./des [e|d] [key] < inputfile > outputfile\n"`** is printed and the program exits with an error code of 1. If the key is too short or long, the error message **`"Key must be exactly 8 bytes\n"`** is printed and the program exits with an error code of 2.

The program shall implement the DES algorithm.

The DES algorithm has two types of inputs and outputs depending on if we are in mode 'e' or mode 'd'. In encryption mode (e), the input is a simple text file written in ASCII, and the output will be an encrypted binary file. In the decryption mode, the input will be a properly encrypted binary file. In both modes, the encryption key is a simple, 8-character ASCII string.

DES is a symmetric cipher, meaning that if you encrypt a file with a given key, you should also be able to decrypt the file with the same key and get the original plaintext.

Design

We have provided a template for you. This template does require additional functions beyond **`main()`**. We have provided the prototypes of some of those functions for you. You will need to finish the prototype functions, which are listed below. You may write additional functions if you feel they are needed, but you MAY NOT modify the provided function headers and you MUST implement functionality in the provided functions.

- **`encrypt_block( unsigned char block[8], unsigned char key[8] )`** - This function will take a single 8-byte block and encrypt it using the 8-byte key.
- **`decrypt_block( unsigned char block[8], unsigned char key[8] )`** - This function will decrypt an 8-byte block using the 8-byte key.

All "magic numbers" that you use must be defined as constants. We have provided the definitions of the magic numbers for you.

Implementation

DES

Please carefully read J. Orlin Grabbe's implementation, linked here. It explains the algorithm, and you are responsible for using the C features required to implement it.

DES is a block cipher, meaning that encryption is done on blocks of 8 bytes at a time.

A block cipher, by itself, has no real encrypting capabilities. It has to be used in a cipher mode of operation in order to establish security. In this assignment, you will be using the simplest (and most insecure) form of block cipher mode, which is the electronic code book (ECB). In ECB, each byte is independently encrypted and decrypted using the same key. This means that if two blocks are the same in the plaintext (such as "POTATOESPOTATOES", two identical 8-byte blocks), then the ciphertext will simply repeat itself twice.

Decryption is identical to encryption, except the 16 keys computed in stage one are used in reverse. Take advantage of this fact so that you don't have to duplicate your code.

We have provided a example of the inputs and outputs for a single block through the 16 rounds of DES that you can use to walk through your code. We recommend using print statements with **`%x`** format specifier to print out the hex values of each byte. The hex values can be directly translated to binary.

Binary Files

Binary files are difficult to examine using tools like nano - one very useful command line tool is **`hexdump -C`**, which makes it possible to see the data of a binary file.

Command Line Arguments

Command line arguments in C require two parameters to the **`main`** function: **`int argc`** and **`char **argv`**.

`argc` is the count of command line arguments including the name of the executable. That means there are three command line arguments expected for a valid run of the **`des`** program (**`./des [e|d] [key]`**).

`argv` is a 2D array of characters that represents the values of the command line arguments (the *s imply it's a pointer to a character pointer, but for this homework you only need to think of the **`argv`** like a 2D array). **`argv[0]`** is the executable (e.g., **`./des`**), **`argv[1]`** is the first command line argument (e.g., **`'d'`** or **`'e'`**), and **`argv[2]`** is the second command line argument (e.g., the key). Another way to think of the argv parameter is as an array of strings. This means that each command line argument string ends in the null character (e.g., **`'\0'`**). So we could model the command line arguments for Test 1 as a 2D array that looks like:

```
'.' '/' 'd' 'e' 's' '\0'

'e' '\0'

'p' 'a' 's' 's' 'w' 'o' 'r' 'd' '\0'
```

That means when implementing the requirements related to checking the command line arguments, you are interested in where the null character is. For the first command line argument that specifies if you're encrypting or decrypting, **`argv[1][1]`** should be `'\0'`. For the second command line argument that specifies the key, **`argv[2][8]`** should be `'\0'`. A null character earlier or later would be an invalid set of command line arguments.

When implementing, we recommend that you start with code that will pass Tests 5, 6, and 7 first and then work on the DES algorithm.

Testing

When compiling your program, you must use the following command:

```
% gcc -Wall -std=c99 des.c -o des
```

OR you may use the provided Makefile:

```
% make clean
% make des
```

Test cases for the program are provided. We have also provided a script, **`test.sh`**, that will execute all of the tests and report the results. There should be no difference between your actual output and the expected outputs. Grading will test additional inputs and outputs, so you should test your program with inputs and outputs beyond the provided example.

The test cases, testing script, template file, and Makefile are provided for you in hw3_starter.tar. You can untar the starter file using the command:

```
% tar xvf hw3_starter.tar
```

```
% cp des_template.c des.c
```

...edit **`des.c`**, compile it, test it, etc...
When you are ready to run the full suite of tests, execute:

```
% chmod +x test.sh
```

```
% ./test.sh
```

Test ID	Mode	Key	Test Input	Exit Code	Expected Output

Test 1	e	password	plaintext_in_1	0	expected_ciphertext_out_1
Test 2	e	password	plaintext_in_2	0	expected_ciphertext_out_2
Test 3	d	candybar	ciphertext_in_1	0	expected_plaintext_out_1
Test 4	e	candybar	ciphertext_in_2	0	expected_plaintext_out_2
Test 5	a	password	ciphertext_in_1	1	expected_error_out_1
Test 6	d	short	ciphertext_in_1	2	expected_error_out_2
Test 7	d	keytoolong	ciphertext_in_1	2	expected_error_out_3

Instructions for Submission

Submit **des.c, Makefile,** and all of the actual outputs generated through testing **(actual_*)** in a **tar** file to **"Homework 03 Submit"** locker through the Moodle Course Website. By submitting the actual results from the tests, you will prove that you have tested your program with the minimum set of the provided acceptance tests.

We recommend that you keep the files for Homework 3 in a separate directory from your other course materials. Assuming that this is so and you're currently in the Homework 3 directory, use the following command to **tar** your files:

```
% make clean
% make
% ./test.sh
```

(Make sure all tests passed - otherwise fix your code)

```
% tar cvf unityid_03_homework.tar des.c Makefile
actual_*
```

This command will **tar** all the specified files (in this case, **des.c, Makefile,** and all of your expected output files of the pattern **actual_***) in the current directory to a file named **unityid_03_homework.tar**. Use your unity id instead of the literal string "unityid". As the **tar** is created, all the files added to the **tar** will be listed so you can confirm that the appropriate files are submitted.

Follow the CSC 230 Style Guidelines. Make sure your program compiles on the common platform cleanly (no errors or warnings), with the required compile options.

There will be a **5 point deduction** for submissions that do not include the **`Makefile`** and ALL actual output files in the tar in addition to **`des.c`**

There will be a **5 point deduction** for submissions that are not tarred or where the tar file is named incorrectly.

There will be a **5 point deduction** for files (**`des.c, Makefile,`** and all of the actual outputs generated through testing **`(actual_*)`**) that are named incorrectly.

There is a 24 hour window for late submissions. After the main locker closes, submit all late work to **"Homework 03 Late Submit."** There is an automatic **20 point deduction** for late work, even if you submit part of the assignment on time.

Rubric

`des.c`

+15 for compiling on the common platform with **`gcc –Wall –std=c99`** options

+70 for passing teaching staff test cases, which demonstrates a correct implementation.

+20 for comments, style, and constants

-5 for meaningless or missing comments

-5 for inconsistent formatting

-5 for magic numbers

-5 for no name in comments

Total: 105 points

Global deductions FROM the score you earn above:

-15 points for any compilation warnings. Your program must compile cleanly! (if it doesn't compile, you will not receive any credit for test cases or compilation)

-70 points for using other standard library functions other than **`getchar()`**, **`putchar()`**, **`printf()`**, and **`exit()`** or for writing code such that the provided tests pass but the program isn't generic enough to handle additional tests.

-5 points for not including the **`Makefile`** and ALL actual output files in the tar in addition to **`des.c`**

-**5 points** for submissions that are not tarred or where the tar file is named incorrectly

-5 points for files (**`des.c, Makefile,`** and all of the actual outputs generated through testing **`(actual_*)`**) that are named incorrectly.

-20 points for late work, even if you submit part of the assignment on time.

You can not receive less than a 0 for this assignment unless an academic integrity violation occurs.