COMPSCI 2211b
Software Tools and Systems Programming

Assignment #4

C Programming Part 2

Posted:	March 23rd 2018
Due:	April 7th 2018 11:55PM
Total:	100 Points (5% of Final Grade)

Learning Outcomes

By completing this assignment, you will gain and demonstrate skills relating to:

- Writing, compiling and running C programs.
- Using and creating functions.
- Using pointers and strings.
- Dynamic memory allocation.
- Creating and using header files.
- Documenting and commenting C source code.

Instructions

For this assignment, an electronic submission is required through OWL. This submission should include the source code for each of the C programs your are required to create (including both the .c and .h files). These files should be named as per the instructions given in each question and end in a .c or .h file extension. The files should be attached individually, and not contained in an archive (e.g. zip, tar, etc.) or other binary format (e.g. they should not be in a Word or PDF document). You can copy your files off the course server using the `scp` command on the lab computers or using a UNIX-Like operation system. On Windows, you can use a program like WinSCP to copy files to/from the course server.

All C programs must work on the course server. Programs will be marked on the course sever and it is expected that students at least test their code on the server before submitting. It will be assumed that the GNU89 C standard (the default for `gcc` on the course server) is being used unless you specifically state otherwise in a comment on the first line of your C source code. Your code must work with the GCC compiler on the course server and you may only use C standards that are supported by this compiler. C++ code or code written in other programming languages will be given 0 marks.

You must document your code. It is expected that you document your C source code with comments. Each program should have comments at the top of the file that state what C standard you are using (if different from the default), any special instructions for compiling/running your code, your name, student number and a short description of what the program does. You are also expected to include in-line comments throughout your code to make each line understandable by anyone who reads it. Lines that are self explanatory or straightforward do not need to be documented.

You will be assessed on the following:

- Completion of each question correctly.
- Providing your source code as required in the above instructions.
- Documenting your code with comments as required in the above instructions.
- Ensuring that your code works on the course server.
- Dividing your code into multiple .c files and header files as required by each question.
- Naming your files correctly and submitting via OWL.

Assignment

For this assignment, you will be creating a simple cryptography library for the Caesar Cipher and Vigenere Cipher as well as a program to test these ciphers.

Your program should be divided into the following files:

- **ciphers.c**: Will contain the code for the encryption and decryption functions for each cipher.
- **ciphers.h**: A header file containing function prototypes for each function in ciphers.c.
- **cipher_main.c**: Contains your main function and code to test your cipher functions.

Your cryptography library should contain the following functions:

```
char * caesar_encrypt(char *plaintext, int key)
```

This function takes a null terminated string, `plaintext`, and integer `key` as arguments. The result is a new dynamically allocated null terminated string containing the `ciphertext` (the result of applying the Caesar cipher on the `plaintext`). Your function should not alter the `plaintext` string in any way.

The Caesar cipher works by shifting each letter in the `plaintext` by the amount given in the key. If this amount would shift the letter past the last letter in the alphabet it is wrapped around to the beginning of the alphabet. For example, 'A' shifted by 5 is 'F', 'T' shifted by 20 is 'N', 'H' shifted by 300 is 'V' and 'D' shifted by -4 is 'Z'.

Your function should convert all letters to uppercase before encrypting and ignore any punctuation, numbers or other nonletter characters (they should be left in the `ciphertext` unencrypted).

The string your function returns must be null terminated, the same length as the `plaintext` string and not take up any more space than required. **The string should be allocated on the heap and not the stack.**

Hint 1: You should use the remainder operator (`%`) to handle the wrapping of the letters. Keep in mind that this operator may work differently then you expect for negative numbers.

Hint 2: Functions like `malloc` or `calloc` might be of use here.

Hint 3: You are allowed to use character and string functions like `toupper` and `strcpy` so long as you meet the other requirements of the function.

```
char * caesar_decrypt(char *ciphertext, int key)
```

This function takes a null terminated string, `ciphertext`, and integer `key` as arguments. The result is a new dynamically allocated null terminated string containing the `plaintext` (the result of decrypting the Caesar cipher using the `key`). Your function should not alter the `ciphertext` string in any way.

You can decrypt the Caesar cipher by simply shifting the letters in the other direction by the amount given in the `key` argument.

Your function should ignore any punctuation, numbers or other nonletter characters (they should be left in the `plaintext` as they are in the `ciphertext`).

The string your function returns must be null terminated, the same length as the plaintext string and not take up any more space than required. **The string should be allocated on the heap and not the stack.**

Hint: You are allowed to call your other functions from this function.

```
char * vigen_encrypt(char *plaintext, char *key)
```

This function takes a null terminated string, `plaintext` and a second null terminated string `key` as arguments. The result is a new, dynamically allocated null terminated string containing the `ciphertext` (the result of apply the Vigenere cipher on the `plaintext`). Your function should not alter the `plaintext` or `key` strings in any way.

The Vigenere cipher works in a similar manner as the Caesar cipher, however, instead of shifting each letter by a fixed amount, the letter is shifted by the letter in the same place in the key. If the `key` is shorter than the `plaintext`, pad the `key` by repeating it.

Example:

Plaintext: `ATTACKATDAWN`
Key: `LEMON`
Padded Key: `LEMONLEMONLE`
Ciphertext: `LXFOPVEFRNHR`

The first letter of the `plaintext` is 'A' and is shifted by 'L', as 'L' is the 11th letter of the alphabet (if we start counting at 0) 'A' is shifted by 11, resulting in 'L'. The second letter of the `plaintext` is 'T' and is shifted by 'E' (the 4th letter of the alphabet) and the result is 'X'. The third letter 'T' is shifted by 12 ('M' is the 12th letter of the alphabet) and wrapped around to 'F'.

Your function should convert all letters in the `plaintext` and `key` to uppercase before encrypting. Ignore any nonletter characters in the `plaintext` (they should be left in the `ciphertext` unencrypted). When a nonletter character is encountered in the `plaintext`, this uses a letter in the `key` even if the character is not encrypted. Assume the `key` contains only letters.

The string your function returns must be null terminated, the same length as the `plaintext` string and not take up any more space than required. **The string should be allocated on the heap and not the stack.**

```
char * vigen_decrypt(char *ciphertext, char *key)
```

This function takes a null terminated string, `ciphertext` and a second null terminated string `key` as arguments. The result is a new dynamically allocated null terminated string containing the `plaintext` (the result of decrypting the `ciphertext` using the Vigenere cipher and `key`). Your function should not alter the `ciphertext` or `key` strings in any way.

You can decrypt the Vigenere cipher by simply shifting the letters in the other direction by the value of the letter in the same place in the `key` argument.

Your function should ignore any punctuation, numbers or other nonletter characters (they should be left in the `plaintext` as they are in the `ciphertext`). Assume the `key` only contains letters.

The string your function returns must be null terminated, the same length as the plaintext string and not take up any more space than required. **The string should be allocated on the heap and not the stack.**

```
void freq_analysis(char *ciphertext, double letters[26])
```

This function takes a null terminated string, `ciphertext` and an array of doubles of size 26. There is no return value, rather your function will be setting the values of the `letters` arrays. Your function should not alter the `ciphertext` string in any way.

The purposes of this function is to perform a frequency analysis of the cipher text. That is, count the number of occurrences of each letter. The result of this analysis should be stored in the `letters` array such that `letters[0]` is the percentage of 'A' or 'a's in the `ciphertext` and `letters[25]` is the percentage of 'Z' or 'z's in the `ciphertext`. The percentage should be a double value between 0 and 100 (inclusive).

You **can not** assume that the `letters` array is initialized to zero. Assume that it may start with any values in the array. Ignore any punctuation, numbers or other nonletter characters. These should not be counted in the total number of characters or in the `letters` array. Uppercase and lowercase letters should be treated as the same letter.

For this function only use pointer notation and not array subscript notation. You should have no '[' or ']' characters in the code for this function after the function header.

Your `main` function

Your `main` function in *cipher_main.c* should ask the user for a line of text to encrypt (that may contain spaces), the encryption cipher to use, and the key to use. Read in these values via standard input using any method you think is appropriate (`getchar`, `scanf`, `gets`, `fgets`, etc.).

You should use the functions from your cryptography library and print the following:

- Display the `plaintext` before encrypting.
- Display the `ciphertext` after encrypting.
- Display the `plaintext` after decrypting.
- Print a nicely formatted table of letters and their frequency in the `ciphertext`.

You should do any needed error checking on the user input including checking for invalid characters in the key (for the Vigenere cipher) and return the correct exit status from your program.

For all of the functions in *ciphers.c* you should output an error and exit the program with a failure exit status if you were unable to allocate memory on the heap.

Ensure that you `free` the dynamically allocated strings before exiting your program.

Other than error messages, only the main function should output text to the screen.

Hint: When reading in the key for the Vigenere cipher, you need to be careful that you do not include the linebreak (\n) on the end of the string. If you use `gets` *or* `fgets` *you will have to manually remove this linebreak before sending the key to the Vigenere functions.*

Example Input/Output

Note that your formatting does not have to match these examples exactly. Your frequency table for example could be split between multiple rows or columns, so long as it is easily readable.

Example 1:

```
Input plaintext:
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Available Ciphers:
1) Caesar
2) Vigenere

Select Cipher: 1

Input key as number: 23

Plaintext:
ABCDEFGHIJKLMNOPQRSTUVWXYZ


Ciphertext:
XYZABCDEFGHIJKLMNOPQRSTUVW


Decrypted plaintext:
ABCDEFGHIJKLMNOPQRSTUVWXYZ


Frequency analysis:
     A    B    C    D    E    F    G    H    I    J    K    L    M    N    O    P    Q    R    S    T    U    V    W    X    Y    Z
   3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8  3.8
```

Example 2:

```
Input plaintext:
The Quick Brown FoX jUMPs Over The Lazy Dog!!

Available Ciphers:
1) Caesar
2) Vigenere

Select Cipher: 1

Input key as number: -3

Plaintext:
The Quick Brown FoX jUMPs Over The Lazy Dog!!


Ciphertext:
QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD!!


Decrypted plaintext:
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG!!


Frequency analysis:
     A    B    C    D    E    F    G    H    I    J    K    L    M    N    O    P    Q    R    S    T    U    V    W    X    Y    Z
   2.9  8.6  2.9  2.9  5.7  2.9  2.9  2.9  2.9  2.9  2.9 11.4  2.9  2.9  5.7  2.9  5.7  5.7  2.9  2.9  2.9  2.9  2.9  2.9  2.9  2.9
```

Example 3:

```
Input plaintext:
The five boxing wizards jump quickly.

Available Ciphers:
1) Caesar
2) Vigenere

Select Cipher: 1

Input key as number: 1337

Plaintext:
The five boxing wizards jump quickly.


Ciphertext:
ESP QTGP MZITYR HTKLCOD UFXA BFTNVWJ.


Decrypted plaintext:
THE FIVE BOXING WIZARDS JUMP QUICKLY.


Frequency analysis:
     A     B     C     D     E     F     G     H     I     J     K     L     M     N     O     P     Q     R     S     T     U     V     W     X     Y     Z
   3.2   3.2   3.2   3.2   3.2   6.5   3.2   3.2   3.2   3.2   3.2   3.2   3.2   3.2   3.2   6.5   3.2   3.2   3.2  12.9   3.2   3.2   3.2   3.2   3.2   3.2
```

Example 4:

```
Input plaintext:
ATTACKATDAWN

Available Ciphers:
1) Caesar
2) Vigenere

Select Cipher: 2

Input key as string:
LEMON

Plaintext:
ATTACKATDAWN


Ciphertext:
LXFOPVEFRNHR


Decrypted plaintext:
ATTACKATDAWN


Frequency analysis:
     A     B     C     D     E     F     G     H     I     J     K     L     M     N     O     P     Q     R     S     T     U     V     W     X     Y     Z
   0.0   0.0   0.0   0.0   8.3  16.7   0.0   8.3   0.0   0.0   0.0   8.3   0.0   8.3   8.3   8.3   0.0  16.7   0.0   0.0   0.0   8.3   0.0   8.3   0.0   0.0
```

Example 5:

```
Input plaintext:
BlewJ's computer quiz favored proxy hacking.

Available Ciphers:
1) Caesar
2) Vigenere

Select Cipher: 2

Input key as string:
DucKs

Plaintext:
BlewJ's computer quiz favored proxy hacking.


Ciphertext:
EFGGB'M MGPJWDWU SEAC HKNRLGN SLQHQ BCMCLHI.


Decrypted plaintext:
BLEWJ'S COMPUTER QUIZ FAVORED PROXY HACKING.


Frequency analysis:
     A     B     C     D     E     F     G     H     I     J     K     L     M     N     O     P     Q     R     S     T     U     V     W     X     Y     Z
   2.7   5.4   8.1   2.7   5.4   2.7  10.8   8.1   2.7   2.7   2.7   8.1   8.1   5.4   0.0   2.7   5.4   2.7   5.4   0.0   2.7   0.0   5.4   0.0   0.0   0.0
```

Example 6:

```
Input plaintext:
The quick onyx goblin jumps over the lazy dwarf.

Available Ciphers:
1) Caesar
2) Vigenere

Select Cipher: 2

Input key as string:
AAAYNAUIIRVHOSVVQRXVRZUONKGFOFPBRHGZCAAAAWIMOWNDUCKS

Plaintext:
The quick onyx goblin jumps over the lazy dwarf.


Ciphertext:
THE DUCKS JUMP BESIDE DIZZY CATS AND LAZY LIONS.


Decrypted plaintext:
THE QUICK ONYX GOBLIN JUMPS OVER THE LAZY DWARF.


Frequency analysis:
     A     B     C     D     E     F     G     H     I     J     K     L     M     N     O     P     Q     R     S     T     U     V     W     X     Y     Z
   7.7   2.6   5.1  10.3   7.7   0.0   0.0   2.6   7.7   2.6   2.6   5.1   2.6   5.1   2.6   2.6   0.0   0.0  10.3   5.1   5.1   0.0   0.0   0.0   5.1   7.7
```

Example 7:

```
Input plaintext:
This is a test!

Available Ciphers:
1) Caesar
2) Vigenere

Select Cipher: 2

Input key as string:
This is a bad key
Error: bad key, invalid char!
```

Example 8:

```
Input plaintext:
Hello World!

Available Ciphers:
1) Caesar
2) Vigenere

Select Cipher: 1

Input key as number: Also a bad key
Error: bad key!
```

Example 9:

```
Input plaintext:
Numb3rs are al0wed in plaintext

Available Ciphers:
1) Caesar
2) Vigenere

Select Cipher: 8
Error: bad selection!
```