Assignment: Encryption and Decryption with Pointers

Learning Outcomes

- Develop familiarity with usage of pointers
- Develop familiarity with handling text files

Task

In cryptography, the Caesar's cipher is a basic encryption technique that for each letter of input that belongs to some alphabet produces an output letter that belongs to the same alphabet but with its position shifted by a known, fixed number of positions. This cryptographic method belongs to a family of *symmetric cryptographic algorithms*: the same ***key*** - the number of position shifts - is used to encrypt plain text into a cipher and to decrypt a cipher into plain text.

The Vigenère cipher improves upon Caesar's cipher by encrypting messages using a sequence of keys which can be represented as a ***keyword***. While a Caesar's cipher adds a fixed value - the key - to each input character during encryption, and requires subtraction of the same value during decryption, a Vigenère cipher's adds the first fixed value - the first letter of a keyword - to the first input character, the second fixed value - the second letter of a keyword - to the second character, and so on, and requires subtraction of the same corresponding values during decryption.

If we wanted to say HELLO [using the ASCII character set] to someone confidentially using Vigenère cipher with a keyword 01 , the encryption process would look as shown below:

Input text	H	E	L	L	O
ASCII code of input text	72	69	76	76	79
Key text	0	1	0	1	0
ASCII code of key text	48	49	48	49	48
ASCII code of cipher text	$72 + 48 = 120$	118	124	125	127
Cipher text	x	v	$\vert$	$\}$	DEL

The ASCII character set includes codes 0 through 127. If the ciphered code goes outside of this range, it should wrap around. For example, if we add character with ASCII code 1 to a character with ASCII code 127, the result should be the character with ASCII code 0; if we subtract, the result should be code 127 again.

In this assignment, you must write a program that:

1. ***Encrypts*** a plain text input file by writing the encrypted text into a ciphered text output file.
2. ***Decrypts*** a ciphered text input file into a deciphered text output file.
3. ***Counts*** the number of words in the deciphered text file; words in the text are delimited by *whitespace characters* [including a space, a new line `'\n'` , a carriage return `'\r'` , a tab `'\t'`]; punctuation characters are assumed to be part of a word.

Two executable programs are required - one executable will encrypt the contents of a plain text input file to a ciphered text output file, while the second executable will decrypt the contents of a [previously encrypted] cipher text input file to a deciphered text output file [which would be exactly similar to the plain text input file provided to the first executable] and also count the number of words in the deciphered text file.

Implementation Details

1. Open a Window command prompt, change your directory to C:\sandbox [create the directory if it doesn't exist], create a sub-directory ass07. Download the zipped archive containing the assignment source files and extract them in this directory. You should have the following files: qdriver.c [partially complete], q.c [partially complete], q.h [partially complete], key.txt, plain.txt, expected-plain.txt, expected-cipher.txt, and expected-output.txt.
2. File key.txt contains the keyword for Vigenère cipher's algorithm; file plain.txt contains the message to be encrypted into file cipher.txt; file expected-cipher.txt represents the correct output of encrypting message file plain.txt; and file expected-plain.txt represents the correct output of decrypting encrypted file cipher.txt [obviously it should be *exactly* the same as the original file plain.txt]. The file expected-output.txt should be exactly similar to expected-plain.txt except for the first line containing the word count in the message [described below].

   > ***Assume the Vigenère cipher's keyword in file key.txt is a single line of characters, doesn't include the `'\n'` character, and that the keyword is composed of a maximum of 255 characters.***
   >
   > ***Do not make any assumptions about the number of characters in plain.txt - it can be an arbitrarily small or an arbitrarily large file.***

3. Declare functions `encrypt` and `decrypt` in q.h [with appropriate function headers] that encrypt and decrypt using Vigenère cipher's algorithm, respectively. These functions would be used by their clients like this:

   ```
   char letter = 'H', keytext[] = "01";
   encrypt(&letter, keytext[0]); // letter is changed to 'x'
   decrypt(&letter, keytext[0]); // letter is changed to 'H'
   ```

4. Define functions `encrypt` and `decrypt` in source file q.c.
5. Study the skeleton code presented in qdriver.c. You'll find the following preprocessor directives:

   ```
   #ifdef ENCRYPT

   // TODO: encrypt plain.txt into cipher.txt

   #else

   // TODO: decrypt cipher.txt into out-plain.txt
   // TODO: write count of words into stderr

   #endif

   ```

If a macro `ENCRYPT` is defined during compilation, only the encryption code is included in the translation unit; otherwise, only the code for decrypting and counting words is included in the translation unit.

> ***A source file, together with its include files, files that are included using the `#include` preprocessor directive, but not including sections of code removed by conditional-compilation directives such as `#ifdef`, is called a translation unit.***

You will not define macro `ENCRYPT` in source file qdriver.c. Instead, you will use an additional gcc compilation option -DENCRYPT [-D followed by the preprocessor macro name you want to define].

6. Implement function `main` in qdriver.c:

    1. Use function `fopen` to open and create appropriate files. For encryption, open input files key.txt and plain.txt, and create output file cipher.txt, while for decryption, open input files key.txt and cipher.txt, and create output file out-plain.txt. Use `fgets` [to read data line-by-line] from an input file stream and `fputs` [to write data line-by-line]. Finally, use `fclose` to close previously created file streams.

        If function `fopen` is unable to create a file stream to or from a file [say plain.txt], your implementation must print the following text to `stderr`

        ```
        Unable to open file plain.txt

        ```

        and exit the program.

    2. Read the keyword from file key.txt into a character array.

    3. Use your previously defined function `encrypt` convert each character read from input file stream [such as plain.txt] and write the encrypted character to an output file stream [such as cipher.txt]. Likewise, use your previously defined function `decrypt` to decrypt each character read from an input file stream [such as cipher.txt] and write the decrypted character to an output file stream [such as out-plain.txt].

        > ***Do not read the entire input file first into a character array before converting characters and writing them into an output file - assume that the input file can have an arbitrarily large size.***

    4. Use function `fprintf` to write the word count during decryption to standard error stream `stderr`. If the number of words in cipher.txt is 789, write the following text to `stderr`:

        ```
        Words: 789

        ```

7. Compile and run both versions [first for encryption, then for decryption] of an executable file like this:

```
$ gcc -std=c11 -pedantic-errors -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -c -o q.o q.c
$ gcc -std=c11 -pedantic-errors -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -DENCRYPT -c -o qdriver.o qdriver.c
$ gcc q.o qdriver.o -o encrypter
$ ./encrypter 1> actual-output.txt 2>> actual-output.txt
$
$ gcc -Wall -Werror -Wextra -Wconversion -Wstrict-prototypes -pedantic-errors -std=c11 -c -o qdriver.o qdriver.c
$ gcc q.o qdriver.o -o decrypter
$ ./decrypter 1>> actual-output.txt 2>> actual-output.txt
$
$ cat out-plain.txt >> actual-output.txt
```

Program `encrypter` will encrypt contents of file plain.txt into file cipher.txt. The cipher.txt created by your program must be identical to file expected-cipher.txt. Any error messages generated by the program are redirected from standard output and standard error streams to file actual-output.txt. This file will be empty for correct versions of program `encrypter`.

Program `decrypter` will decrypt contents of cipher.txt [which was earlier created by program `encrypter`] into file out-plain.txt and further write a message [specifying the word count in file cipher.txt] to the standard error stream [which is being appended to file actual-output.txt]. File out-plain.txt created by your program must be identical to file plain.txt and expected-plain.txt.

Finally, the Linux shell command `cat` appends the contents of file out-plain.txt to the contents of file actual-output.txt. You can compare your actual-output.txt with the correct output in expected-output.txt.

8. Use command diff to compare your files with the correct versions to ensure your submission is correct:

```
$ diff --strip-trailing-cr expected-plain.txt plain.txt
$ diff --strip-trailing-cr expected-plain.txt out-plain.txt
$ diff --strip-trailing-cr expected-cipher.txt cipher.txt
$ diff --strip-trailing-cr expected-output.txt actual-output.txt
```

9. Finally test your code with a variety of keywords and plain text files.

File-level and Function-level documentation

Every source and header file you submit *must* contain file-level documentation blocks whose purpose is to provide human readers [yourself and other programmers] useful information about the purpose of this source file at some later point of time

Every function that you declare in a header file [and define in a corresponding source file] must contain a function-level documentation block.

> ***Don't copy and paste documentation blocks from previous assignments. Annoyed graders will definitely subtract grades to the full extent specified in the rubrics below when they detect such copy-and-paste scenarios.***

Submission and automatic evaluation

1. In the course web page, click on the submission page to submit the necessary files.
2. Read the following rubrics to maximize your grade. Your submission will receive:
    1. F grade if your submission doesn't compile with the full suite of `gcc` options [shown above].
    2. F grade if your submission doesn't link to create an executable.
    3. $A+$ grade if the submission's output matches the correct output. Otherwise, a proportional grade is assigned based on how many incorrect results were generated by your submission.
    4. A deduction of one letter grade for each missing documentation block. Every submitted file must have one file-level documentation block. Every function that you declare in a header file must provide a function-level documentation block. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and the three documentation blocks are missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the three documentation blocks are missing, your grade will be later reduced from C to F.