CIS241 Project One

Substitution Ciphers

(Due Feb. 22, 2019)

1. Problem Statement

Caesar ciphers are a simple form of substitution ciphers in which each letter in the original text is replaced by a letter some fixed number of positions down the alphabet. Such a fixed number is referred to as the key. Obviously, it is easy to perform, but also easy to crack - just try out all 25 keys. A better version of substitution ciphers is to use a word as the key, rather than a number.

Suppose the key word is FEATHER. Then you first remove duplicate letters, yielding FEATHR, and append the other letters of the alphabet in reverse order:

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

Now encryption can be described as a mapping shown below:

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

Here, the upper row represents each of the original letters in the text and the lower row describes the corresponding cipher letter in the encrypted text. As such, encrypting a file is to substitute a cipher letter for each letter in the file as shown above and decrypting an encrypted file is just to do the opposite.

Note that the above example shows only how to process uppercase letters. Similarly you should process lowercase letters in a similar fashion; for example, substituting letter ‘f’ for letter ‘a’ on the basis of the key given above. For simplicity, you can assume the user enters a key in all lowercase.

2. Program Requirements

Write a program that encrypts or decrypts a file using this version of substitution ciphers. Your program should provide the user with two options, encryption and decryption. It should also allow the user to specify the key word as well as the input and output filenames. Use command line arguments for the user to pass them into your program.

If the user chooses to do encryption, your program will produce an output file containing text that is encrypted with a key the user provides. And then if the user runs your program again for decryption, your program should be able to use the same key (provided by the user) to decrypt the encrypted file and produce an output file that is identical to the original file before encryption is done.

Specifically, you can use the following command line to compile your program.

```
gcc proj1.c -o proj1
```

If your program compiles successfully, you can use a command line as shown below to run your program, where option 1 is for encryption, “textbook” is the key word, “original.txt” is the input filename, and “encrypted.txt” is the output filename.

```
./proj1 1 textbook original.txt encrypted.txt
```

Then you can use a command line as shown below to run your program again, where option 2 is for decryption, "textbook" is the key word, "encrypted.txt" is the input filename, and "decrypted.txt" is the output filename.

```
./proj1 2 textbook encrypted.txt decrypted.txt
```

If your program works correctly, the texts in files "originall.txt" and "decrypted.txt" should be identical.

3. Program Structure

Note that the mapping shown above is to give you an idea as to how encryption is done; it does not imply that you have to use two arrays to do encryption. Actually, one array that holds the letters as shown in the lower row will be sufficient. As shown above, letters in the upper row are in alphabetic order, from 'A' to 'Z' (that is, from 65 to 90 in their integer form); thus, each letter's position relative to letter 'A" is the index of the corresponding cipher letter in the lower row. Since the relative position of each original letter can be calculated, an array for the upper row is not needed. Similarly, you can use one array to do decryption – find a way to create such an array in order to perform decryption. In other words, no linear search for either encryption or decryption is necessary. Calculating the index of a substitute letter, rather than using linear search, allows your program to do the work in the most efficient way. Hence, declare two arrays as follows in your main function:

```
char encrypt[26], decrypt[26];
```

Also define the following functions and use them in your program. Each function represents a step in your program to achieve its goals, except for function *found()* which is a helper for other functions.

```
// search the first n characters in array list for character target
// return true if it is found, and false otherwise
int found(char list[], int n, char target);

// initialize the encrypt array with appropriate cipher letters according to the given key
void initializeEncryptArray(char key[], char encrypt[]);

// initialize the decrypt array with appropriate substitute letters based on the encrypt array
void initializeDecryptArray(char encypt[], char decrypt[]);

// process text from the input file and write the result to the output file
// pass the encrypt array to parameter substitute if encryption is intended
// pass the decrypt array to parameter substitute if decryption is intended
void processInput(FILE * inf, FILE * outf, char substitute[]);
```

Since each function above represents a step for your program to do its work, you can develop the program in an incremental fashion; that is, define one function, test it, and if it works correctly, move on to the next function. A minor error may produce a quite different result for a program like this. Incremental development is an effective way to ensure a correct program.

When you test your program, you need to run it to generate an encrypted file from an input file, run your program again with the encrypted file as input to produce a decrypted file, and then compare the decrypted file to the original data file and see if they are identical. Since such a process involves multiple command lines and you have to run them as often as needed, it would be much convenient to define a *makefile* and use command *make* to automate your testing process. Also use Linux command *diff –s original.txt decrypted.txt* to see if they are identical. You can add this command line to your *makefile* so as to perform your test and compare the test results automatically.

4. Submission Requirements

When your program works correctly, make a Word or PDF file with your source code, *makefile*, and two sets of test data and program outputs (one for encryption and one for decryption), and submit it via Blackboard.