1 Instructions

You must work **alone** on this project. Your project solution document must be **uploaded** to **Blackboard** by the assignment **deadline**. **Section 6** describes what to submit and by when; read it now.

2 Lab Project Objectives

1. Write a basic C program.
2. Define and use local/global variables and local/global constants.
3. Write and call static and nonstatic functions and pass parameters by value.
4. Define and use C-string arrays.
5. Use flow of control constructs such as if-statements, for-loops, and while-loops.
6. Parse the command line for arguments and options.
7. Perform text file I/O. Use printf() for output.
8. Define and use struct variables and define new types using the typedef statement.
9. Write preprocessor macros with #define.
10. Write preprocessor guards with #ifndef … #endif.
11. Understand the difference between declaration and definition.
12. Understand the difference between static and nonstatic functions, variables, and constants.
13. Use the GCC C compiler to compile a C program. Use the GDB debugger to debug a program.
14. Perform automated testing using a Bash shell script.

3 Vigenère Encryption

Cryptography is the process of **encrypting** messages—so they cannot be read by an interceptor—and **decrypting** encrypted messages. A message to be encrypted is referred to as the **plaintext** and the encrypted message is referred to as the **ciphertext**. The plaintext is encrypted using a secret **key** and if only the encryptor and decryptor know the key, then the message can be securely transmitted[1].

Cryptanalysis is the process of attempting to decrypt an encrypted message when you do not know the key. A **strong** encryption algorithm is one that is considered to be highly resistant to a cryptanalytic attack. Strong encryption algorithms are difficult to design, and the current gold standard is the **Advanced Encryption Standard** (AES). The AES is a complicated algorithm to implement, so we will not be discussing it.

Historically, encryption has been used since ancient times, especially by militaries and governments. All classical encryption algorithms[2] are weak in that they are especially vulnerable to cryptanalysis and are therefore not secure. But, we won't let us stop that from having some fun.

The Vigenère cipher was originally developed in 1553 by Giovan Battista Bellaso, an Italian lawyer and cryptographer. In the 19th century, the cipher was misattributed to Blaise de Vigenère, a French diplomat who, in 1568, described a minor variation of the Bellaso cipher. The Vigenère cipher was easy to implement but was notably difficult—at that time—to break, being given the description, *le chiffre indéchiffrable* (the indecipherable cipher).

The Vigenère cipher is based on a tabula recta (rectangular table) which consists of the letters of the alphabet being used to write and encrypt the message. We will make use of the 26-letter English alphabet. Our English language tabular recta is shown on the next page.

Next, a key is chosen. Ideally, the key should be as long as the plaintext, but a shorter, easier to remember key can be chosen. Let's suppose our plaintext is "ATTACKATDAWN". First, notice that the message is written in all-uppercase letters and no spaces or punctuation marks are permitted (doing so would make the ciphertext more vulnerable to cryptanalysis). Let's chose "TANGO" as the key. To encrypt, we write the key above the plaintext, repeating the letters of the key to be as long as the plaintext,

```
TANGOTANGOTA
ATTACKATDAWN
```

Vigenère Tabula Recta for English Text

	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

[1] If the same key is used for both encryption and decryption, then the key is **symmetric**. A nonsymmetric cipher would be one where different keys are used for encryption and decryption. The well-known RSA encryption algorithm is a nonsymmetric cipher.

[2] Those invented and in wide use prior to the invention of computers.

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

We start with the first letter of the key being T and the first letter of the plaintext being A. Go to row T in the table and column A, and write the letter that appears in the table at the intersection of that row and column,

```
TANGOTANGOTA
ATTACKATDAWN
T
```

Now, move one letter to the right and repeat. We navigate to row A and column T and write the letter that appears at the intersection of that row and column,

```
TANGOTANGOTA
ATTACKATDAWN
TT
```

Next, row N and column T,

```
TANGOTANGOTA
ATTACKATDAWN
TTG
```

Finishing the encryption,

```
TANGOTANGOTA
ATTACKATDAWN
```

```
TTGGQDAGJOPN
```

Therefore, the ciphertext is TTGGQDAGJOPN. The main idea behind the Vigenère cipher is to disguise plaintext **letter frequencies** which makes the ciphertext vulnerable to cryptanalysis. The Vigenère cipher is an example of a **polyalphabetic cipher** which means that the same letter in the plaintext may not always (and should not always) be encrypted as the the same letter in the ciphertext (as would be the case in a **monoalphabetic cipher**). For example, in a monoalphabetic cipher, if the most common letter is P, then it can be deduced (perhaps incorrectly, but perhaps not), that P in the ciphertext must correspond to the letter E in the plaintext since E is the most commonly used letter in the English alphabet. A is also a very common letter in English text, but in the Vigenère cipher of "ATTACKATDAWN" you can see that the letter A was encrypted as T, G, A, and O.

The main cryptanalytic attack against the Vigenère cipher is to attempt to deduce the length of the key by noting repeated occurrences of letters or combinations of letters in the ciphertext. For example, if the plaintext is encrypted to become the ciphertext "HINWBOLBHIJWEBJ", you may notice that the subsequence HI appears in positions 1 and 9. The distance between the first and second occurrence is 8. An attacker might guess that the length of the key, then, is 8 or some factor or multiple of 8 such as 1, 2, 4, 16, or 24. Once the attacker has a good guess of the length of the correct key, then the cipher is vulnerable to cryptanalysis.

In closing, we note that if the key is as long as the plaintext, the letters in the key are truly random, and the key is never reused, then the Vigenère cipher is theoretically unbreakable because all cryptanalytic attacks depend on finding and identifying letter frequencies.

4 Lab Project

Your lab project is to write a complete C program which implement Vigenère encryption. The program shall retrieve arguments and options from the command line. The synopsis of the command is,

```
vigenere mode [-h] -k keyfile [-v]
```

Where *mode* is **e** for encryption or **d** for decryption. The -k option is followed by the name of the text file containing the key. The optional -h option displays a help message, e.g.,

```
$ vigenere -h
Vigenere Cipher Version 1.0

Encrypts or decrypts a message using the Vigenere cipher.

Usage: vigenere mode [-h] -k keyfile [-v]

If performing encryption (mode = e), the plaintext is read from stdin and the ciphertext is
written to stdout. If performing decryption (mode = d), the ciphertext is read from stdin and
the plaintext is written to stdout. Modes are:

    e  Encrypt the plaintext to produce the ciphertext using the specified key.
    d  Decrypt the ciphertext to produce the plaintext using the specified key.

Options:
    -h  Displays this help message and terminates without further processing.
    -k  Reads the key from 'keyfile'.
    -v  Displays version info and terminates without further processing.
```

The -v option displays version information, e.g.,

```
$ vigenere -v
Vigenere Cipher Version 1.0
```

If *mode* is missing, the program shall display an error message, e.g.,

```
$ vigenere -k key.txt
vigenere: missing mode (should be 'e' to encrypt or 'd' to decrypt).
```

If -k and/or *keyfile* is/are missing, the program shall display an error message, e.g.,

```
$ vigenere e
vigenere: missing -k 'keyfile' option. Use -h option for help.
$ vigenere e -k
vigenere: -k option, missing key file name.
$ vigenere e Key.txt
vigenere: invalid command line option: Key.txt
```

The program shall read the plaintext from **stdin** and shall write the ciphertext to **stdout**, e.g.,

```
$ cat > Key.txt
TANGO^D
$ vigenere e -k Key.txt
ATTACKATDAWN^DTTGGQDAGJOPN
```

If you wish, you may redirect **stdin** and/or **stdout**, e.g.,

```
$ cat > Plain.txt
ATTACKATDAWN^D
$ vigenere e -k Key.txt < Plain.txt > Cipher.txt
$ cat Cipher.txt
TTGGQDAGJOPN
```

Note, *mode* does not have to be the first argument,

```
$ vigenere -k Key.txt e < Plain.txt > cipher.txt        works just fine
```

4.1 Program Design

Because **software design** is one of the most difficult-to-learn skills in programming, I am going to give you a template for the program. Navigate to the course website and download the **Extra Credit Project 1 Tarball**. Extract the tarball to a working directory, e.g.,

```
~ $ tar xjf cse220-s14-e01.tar.bz2      files will be extracted to cse220-e01
~ $ cd cse220-e01
~/cse220-e01 $ ls
Controller.c    File.c    Globals.c    Main.c    Makefile    Model.h     String.h    View.c    Vigenere.c
testcases
Controller.h    File.h    Globals.h    Main.h    Model.c     String.c    Types.h     View.h    Vigenere.h
test.sh
```

The code downloaded from blackboard is written by another author and as such their author information is included in the comments for each file. When you modify these files, you must comment on your own modification history in the appropriate place. **You must not replace the original author's information with your own information.**

This program uses the Model/View/Controller (MVC) architectural pattern[3] to separate the storage of data from the user interface. In the MVC architecture, the View provides the user interface (display and reading input). The Model stores data which is used by the View in interacting with the user. The Controller acts as an intermediary between the View and the Model.

Your job is to finish the code by writing the necessary code in the places indicated by ??? symbols. To properly understand the code, you will need to read the comments and trace through the various functions to see how the different modules (Controller, File, Model, View, and Vigenere) interact.

4.2 Avoiding Creating the Tabula Recta

When I first started writing the code for this project, I assumed we would have to create a two dimensional array and fill it up with the alphabetic letters as shown in the tabula recta. However, it soon became apparent to me that we don't have to do that. You can derive a relationship among the alphabet, the row, and the column of the table during the encryption/decryption process. To see this, let's take our earlier example,

```
TANGOTANGOTA
ATTACKATDAWN
```

Let i be the index of the character in the plaintext that is currently being used for encryption, $i = 0, 1, 2, \ldots, strlen(plaintext)-1$. Let $plaintext_i$ be the character at index i that is being encrypted and let key_i be the corresponding key character.

$i = 0$, $plaintext_i$ = 'A', key_i = 'T', $col = plaintext_i$ – 'A' = 'A' – 'A' = 0, $row = key_i$ – 'A' = 'T' – 'A' = 19

If we go to row 19, column 0 in the tabula recta, we see that character at that spot is 'T'. Thus,

$ciphertext_i$ = 'A' + (row + col) % 26 = 'A' + (19 + 0) % 26 = 'A' + 19 = 'T'

Next,

$i = 1$, $plaintext_i$ = '*T*', key_i = 'A', $col = plaintext_i$ – 'A' = '*T*' – '*A*' = 19, $row = key_i$ – 'A' = '*A*' – '*A*' = 0

If we go to row 0, column 19 in the tabula recta, we see that character at that spot is 'T'. Thus,

$ciphertext_i$ = 'A' + (row + col) % 26 = 'A' + (0 + 19) % 26 = 'A' + 19 = 'T'

Next,

$i = 2$, $plaintext_i$ = '*T*', key_i = 'N', $col = plaintext_i$ – 'A' = '*T*' – '*A*' = 19, $row = key_i$ – 'A' = '*N*' – '*A*' = 13

If we go to row 13, column 19 in the tabula recta, we see that character at that spot is 'G'. Thus,

$ciphertext_i$ = 'A' + (row + col) % 26 = 'A' + (13 + 19) % 26 = 'A' + 6 = 'G'

So, I hope you see the pattern for determining $ciphertext_i$ from $plaintext_i$ and key_i,

Function *EncryptChar* (**Input:** *plaintext*, *key*, *i*) **Returns** $ciphertext_i$
    $row \leftarrow key_i$ – 'A'
    $col \leftarrow plaintext_i$ - 'A'
    $ciphertext_i \leftarrow$ 'A' + (row + col) % 26
End Function

[3] http://en.wikipedia.org/wiki/Model-view-controller

Now, what about decrypting?

i = 0: $ciphertext_i$ = '*T*', key_i = 'T', $col = ciphertext_i$ – 'A' = 'T′ – *'A′* = 19, $row = key_i$ – 'A' = 'T′ – *'A′* = 19

$plaintext_i$ = 'A' + (*col* – *row* + 26) % 26 = 'A' + (19 – 19 + 26) % 26 = 'A' + 0 = 'A'

Next,

i = 1: $ciphertext_i$ = '*T*', key_i = 'A', $col = ciphertext_i$ – 'A' = 'T′ – *'A′* = 19, $row = key_i$ – 'A' = 'A′ – *'A′* = 0

$plaintext_i$ = 'A' + (*col* – *row* + 26) % 26 = 'A' + (19 – 0 + 26) % 26 = 'A' + 19 = 'T'

Next,

i = 2: $ciphertext_i$ = '*G*', key_i = 'N', $col = ciphertext_i$ – 'A' = 'G′ – *'A′* = 6, $row = key_i$ – 'A' = 'N′ – *'A′* = 14

$plaintext_i$ = 'A' + (*col* – *row* + 26) % 26 = 'A' + (6 – 14 + 26) % 26 = 'A' + 18 = 'T'

And so on, so the pattern is,

Function *DecryptChar* (**Input:** *ciphertext*, *key*, *i*) **Returns** $plaintext_i$
    row ← key_i – 'A'
    col ← $ciphertext_i$ - 'A'
    $plaintext_i$ ← 'A' + (*col* - *row* + 26) % 26
End Function

5 Testing

No project is complete without adequate testing. For this project, I have given you several test cases (located in the **testcases** subdirectory). There are four test cases in the triplets of files (key1.txt, plain1. txt, cipher1.correct), (key2.txt, plain2.txt, cipher2.correct), (key3.txt, plain3.txt, cipher3.correct), and (key4.txt, plain4.txt, cipher4.correct).

You can test your program manually using one of these test cases by performing these commands,

```
~/cse220-p02 $ make                                    program builds successfully with no errors
~/cse220-p02 $ cp vigenere testcases                    copy binary to the testcases subdirectory
~/cse220-p02 $ cd testcases                                make the testcases subdirectory your current directory
~/cse220-p02/testcases $ ./vigenere e -k key1.txt < plain1.txt > cipher1.txt
~/cse220-p02/testcases $ diff cipher1.txt cipher1.correct
~/cse220-p02/testcases $ ./vigenere d -k key1.txt < cipher1.txt > plain2.txt
~/cse220-p02/testcases $ diff plain1.txt plain2.txt
```

In this test case, we encrypt the message in **plain1.txt** using the key stored in **key1.txt** and send the ciphertext to a file named **cipher1.txt**. Then we compare the encrypted message in **cipher1.txt** to the correct encrypted message (generated by my bug-free program) stored in **cipher1.correct**. If the program is working correctly then **cipher1.txt** and **cipher1.correct** should be identical The **diff** command (short for “difference”) will display the differences between two files, so we use **diff** to compare **cipher1.txt** and **cipher1.correct** to determine if there are any differences between them. If there are not (because the files are identical), then **diff** will not display anything. Otherwise, **diff** will display the lines from the files that differ. If **diff** outputs nothing, then you can say that your program passed that **encryption** test case.

To test **decryption** (which assumes the encryption passed) then we decrypt the message in **cipher1.txt** using the key stored in **key1.txt** and send the plaintext to a file named **plain2.txt**. If the program is working correctly then **plain1.txt** and **plain2.txt** should be identical so we use **diff** to compare them. If there are no differences, then we can say that the program passed that **decryption** test case.

To automate this process, I have provided a **Bash shell script** in the project directory named **test.sh**. This shell script will perform the following operations:

1. It will copy the binary **vigenere** to the **testcases** subdirectory.
2. It will **cd** to the **testcases** subdirectory.
3. It will perform the last four commands shown above on each of the four pairs of test case files.
4. It will then **cd** back to the project directory.

To execute the shell script, type these commands,

```
~/cse220-p02 $ make                    program builds successfully with no errors
~/cse220-p02 $ ./test.sh               run Bash shell script to automate the testing
```

If Bash gives you an error message saying that it cannot execute **test.sh** it may be that you need to set the **x** (execute) permission attribute on the file,

```
~/cse220-p02 $ chmod +x test.sh        set the x permission attribute
```

6 What to Submit for Grading and the Assignment Deadline

When your program is complete, type the following commands to "clean" the source code directory and create a bzipped tarball named **cse220-s14-e01-*lastname*.tar.bz2,**

```
~ $ cd cse220-e01
~/cse220-p02 $ make clean
~/cse220-p02 $ cd ..
~ $ tar cvjf cse220-s14-e01-lastname.tar.bz2 cse220-e01
```

Where *lastname* is your surname (or your first name if you do not have a surname).

Submit a working make file so the TA can build and test your project by typing **make clean** followed by **make**.

Submit this tarball to Blackboard using the lab project submission link before the deadline. The **deadline is 4:00am Mon 17 Mar** (that's four o'clock in the morning on Wed, or very late Tue night for those of you who will be attempting to pull an all-nighter). **Consult the online syllabus for the academic integrity policies. This assignment will not be accepted late, and is not subject to any additional extra credit.**