1 Instructions

You may work in **pairs** (that is, as a **group of two**) with a **partner** on this lab project if you **wish** or you may work **alone**. If you work with a partner, only submit **one** lab project with **both** of your **names** in the **source code file** to Blackboard for grading; you will each earn the **same** number of points. **What** to hand in, and **by when**, is discussed in **Section 5**; read it.

2 Lab Objectives

After completing this assignment the student should be able to,

- Complete all of the objectives of the previous lab projects.
- **Read** and **write** data from and to **text files**.
- Write an **end of file loop** to read data from a text file.
- Write a **vary loop** to access the elements of an array.
- Define and use **array** variables.
- Understand how **C-strings** are represented and manipulate the characters of a C-string.

3 Prelab

- Download the *Lab 11.cpp template* from the course website to your Desktop or a temporary folder, e.g., *C:\Temp*. Create a new Code::Blocks project named *Lab11* and add this source code file to your project. Follow the instructions in Steps 28 through 32 of the Code::Blocks tutorial[1] to add the source code file to your project.
- You need to create the plain text file containing the secret message. Create a text file named *plain.txt* containing a super secret message and save the file in the appropriate Code::Blocks folder where it can be opened for reading. Here is an example file containing three plaintext messages, which contain all the letters of the alphabet.

  ```
  THEQUICKREDFOX
  JUMPEDOVER
  THELAZYBROWNDOG
  ```

- You are now ready to proceed to the lab exercise in Section 4. There are no other prelab exercises this week.

4 Lab Exercise

Encryption is the process of converting information from one form to another so as to make it unreadable. For example, we may wish to encrypt a message "Günther is loose again; release the hounds." into an encrypted message "awiweka sliwoflek mzpw odfk sl jwu dj". The unencrypted message is called the **plaintext** and the encrypted message is called the **ciphertext.**

There have been many **encryption algorithms**[2] developed over the last couple of millennia. Some are better than others; the **strength** of an encryption algorithm is a measure of how difficult it is to decrypt a message encrypted using that algorithm without knowing the actual encryption algorithm. That is, if Bob uses secret encryption algorithm *E* to encrypt a plaintext message *P* forming a ciphertext message *C*, and if Alice doesn't know how *E* works, then if *E* is a strong encryption method, it will be extremely difficult—if not impossible—for Alice to decrypt *C* so she can read *P*.

Encryption has obvious military and state security uses and that is what it was originally used for, and still is. Furthermore, today, encryption is widely used on the internet so customers can log in to secure web sites and transmit confidential information such as addresses, phone numbers, and most importantly, credit card numbers. It is an important and interesting subfield of both mathematics and computer science (one of many areas where mathematics and computer science overlap).

Strong encryption algorithms are very complex and very difficult to design. One of the best and well-known strong encryption algorithms is the **Advanced Encryption Standard** (**AES**) endorsed by the National Institute of Standards and Technology (NIST) and the National Security Agency[3] (NSA). Weak encryption algorithms, on the other hand, are quite easy to devise.

For example, we will discuss a simple and very weak algorithm which is just strong enough to mystify your little brother or sister. Our character set for the plaintext and ciphertext messages is the uppercase letters 'A' - 'Z'. The algorithm uses four character arrays named *plain*, *sub_even*, *sub_odd*, and *cipher*. The number of elements in *plain* must be large enough to store the longest plaintext message we can encrypt; let's define that to be 50. Note that if the message being encrypted is less than 50 characters, there will be some array elements that are unused. The ciphertext is written to *cipher* so it needs to be the same size as *plain*, i.e., 50. The sizes of *sub_even* and *sub_odd* will be 26 each, one character for each letter of our character set.

Let's take a concrete example so I can explain how the algorithm works. First, we need to randomly place each character of our 'A' - 'Z' character set into *sub_even* and *sub_odd*. One such ordering is shown on the next page.

1 http://devlang.com/cse100_codeblocks
2 An algorithm is a step-by-step procedure for performing some task or operation. An algorithm can be documented in pseudocode or a flowchart.
3 http://en.wikipedia.org/wiki/Advanced_Encryption_Standard. NSA is the U.S. federal security agency which snoops on most all global communications (Internet, phone, email, satellite, etc) and some domestic U.S. communications.

```
char sub_even[] = {
    'K', 'H', 'J', 'N', 'O', 'L', 'M', 'P', 'S', 'Q', 'R', 'V', 'W', 'T', 'U', 'Z',
    'A', 'X', 'Y', 'C', 'F', 'G', 'E', 'I', 'B', 'D'
};
char sub_odd[]  = {
    'P', 'R', 'V', 'X', 'S', 'U', 'B', 'D', 'Y', 'A', 'E', 'J', 'F', 'I', 'G', 'H',
    'K', 'O', 'N', 'M', 'Q', 'L', 'W', 'T', 'C', 'Z'
};
```

Note when defining an array and initializing it at the same time, we do not need to specify the size of the array in the brackets; the compiler will count how many values are in the initializer list, and will allocate that much room for the array. Both *sub_even* and *sub_odd* are of size 26 since there are 26 characters in the initializer lists.

The *plain* and *cipher* arrays will be defined and allocated of size 50. Each will store a C-string, i.e., each element stores one character of the C-string, and the final character is followed by the null character '\0'. The null character is not part of the string, but rather, tells us where the string ends. For example, the plaintext message "KILLTHEAMBASSADOR" would be stored in *plain* as,

```
char plain[] = { 'K', 'I', 'L', 'L', 'T', 'H', 'E', 'A', 'M', 'B', 'A', 'S', 'S', 'A', 'D', 'O', 'R', '\0' };
```

The algorithm works as follows,

1. We will walk through *plain* encrypting one character at a time. Define an int variable *index* initialized to 0. *index* is an index variable into the array and is the index of the character that is currently being encrypted.
2. Let *plain_char* be the character in *plain* at index *index*, i.e., the character at *plain*[*index*]. The ASCII values of the uppercase letters begin at 65 for 'A', 66 for 'B', 67 for 'C', ..., and 90 for 'Z', in order. We subtract 65 or 'A' from *plain_char* to form another array index named *sub_index* which will be in [0, 25].
3. If *index* is even, we retrieve the character from *sub_even* that is at index *sub_index*. If *index* is odd, we retrieve the character from *sub_odd* that is at index *sub_index*. Call this character *cipher_char*.
4. We write *cipher_char* to the *cipher* array at index *index*.
5. Increment *index*.
6. Repeat 2-6 until we reach the null character in the *plain* array.
7. Write the null character to *cipher* following the last valid character to make *cipher* a proper C-string.

For example, with *plain* as defined above,

Step 1. Initialize *index* to 0.
Step 2. We get the character at *plain*[*index*] calling it *plain_char*. *plain_char* will be 'K'. We subtract 65 or 'A' from *plain_char* to form *sub_index* which will be 10 since the ASCII value of 'K' is 75.
Step 3. Since *index* = 0 is even, we retrieve the character from *sub_even*[*sub_index*] which will be 'R', storing this character in *cipher_char*.
Step 4. We store *cipher_char* in *cipher*[*index*].
Step 5. We increment *index* to 1. At this time, *cipher* contains the character sequence "R".
Step 6. We go to Step 2.
Step 2. We get the character at *plain*[*index*] calling it *plain_char*. *plain_char* will be 'I'. We subtract 65 or 'A' from *plain_char* to form *sub_index* which will be 8 since the ASCII value of 'I' is 73.
Step 3. Since *index* = 1 is odd, we retrieve the character from *sub_odd*[*sub_index*] which will be 'Y', storing this character in *cipher_char*.
Step 4. We store *cipher_char* in *cipher*[*index*].
Step 5. We increment *index* to 2. At this time, *cipher* contains the character sequence "RY".
Step 6. We go to Step 2.
Step 2. We get the character at *plain*[*index*] calling it *plain_char*. *plain_char* will be 'L'. We subtract 65 or 'A' from *plain_char* to form *sub_index* which will be 11 since the ASCII value of 'K' is 76.
Step 3. Since *index* = 2 is even, we retrieve the character from *sub_even*[*sub_index*] which will be 'V', storing this character in *cipher_char*.
Step 4. We store *cipher_char* in *cipher*[*index*].
Step 5. We increment *index* to 3. At this time, *cipher* contains the character sequence "RYV".
Step 6. We go to Step 2.
...
Step 7. We write the null character *cipher*[*index*] to make it a proper C-string. At this time, *cipher* contains "RYVJCDOPWRKNYPN GX" which is our encrypted message. Pretty unreadable huh?

You should trace through the algorithm until you understand it. By the way, this type of encryption algorithm is known as a **substitution cipher** because we are simply substituting one symbol in the plaintext with a different symbol in the ciphertext. Substitution ciphers have been obsolete for centuries due to the ease of breaking them.

For this programming project, you will complete the code I have given you in *Lab11.cpp* by reading the comments and implementing the pseudocode. The program opens an input file named "plain.txt" which contains one of more lines of text containing top secret messages. Note that the messages can only consist of uppercase letters; do not insert spaces, digits, or punctuation. Each plaintext message from "plain.txt" will be read, encrypted, and the corresponding ciphertext message will be written to "cipher.txt", which will be created in the same folder as "plain.txt". After running your program, open "cipher.txt" to verify that the ciphertext messages are correct.

For testing, you can put these three plaintext messages in "plain.txt",

```
THEQUICKREDFOX
JUMPEDOVER
THELAZYBROWNDOG
```

After encrypting each message, the contents of "cipher.txt" should be,

```
CDOKFYJEXSNUUT
QQWHOXULOO
CDOJKZBRXGEINGM
```

Of course, you should test your program on additional test cases that you create.

4.1 Additional Programming Requirements

1. Modify the **comment header block** of the source code file with your name, email address, lab date/time, and your lab TA.
2. Carefully **format** your code and follow the **indentation** of the text as shown in the example programs of the textbook.

5 What to Submit for Grading and by When

When you are finished with the program, upload *Lab11.cpp* to Blackboard using the lab submission link by the deadline. If your program does not compile or run correctly, upload what you have completed for grading anyway (you will generally receive some partial credit for effort). **The deadline for the complete lab project is 4:00am Wed 9 Dec. This is a hard deadline, i.e., lab projects will not be accepted after that time for late penalty points. You may still earn 20% and 10% bonus pts for early submissions.** Consult the online syllabus for the late and academic integrity policies.

6 Grading Rubric

1. Lab Exercise Program (0 to 5 pts)

Testing the Program: Compile and run the student's program on this file, named "plain.txt",

```
THEQUICKREDFOX
JUMPEDOVER
THELAZYBROWNDOG
KILLTHEAMBASSADOR
GUNTHERISLOOSEAGAINRELEASETHEHOUNDS
THEPACKAGEWILLBEONTHELASTFERRY
```

The program should create a file named "cipher.txt" which contains this text,

```
CDOKFYJEXSNUUT
QQWHOXULOO
CDOJKZBRXGEINGM
RYVJCDOPWRKNYPNGX
MQTMPSXYYJUGYSKBKYTOOJOPYSCDODUQTXY
CDOHKVRPMSEYVJHSUICDOJKNCUOOXC
```

Assign pts per the following,

a. If the submitted program does not compile and the student completed practically none of it, assign **+0 pts**.
b. If the submitted program does, or does not, compile and the student completed less than 50% of the required code correctly, assign **+2 pts**.
c. If the submitted program does not compile and the student completed more than 50% of the required code correctly, assign **+3 pts.**
d. If the submitted program compiles and the student completed more than 50% of the required code correctly, assign **+4 pts**.
e. If the submitted program compiles and is implemented perfectly, or close to perfect with only one or two minor mistakes, assign **+5 pts**.

3. Deadline was 4:00am Wed 9 Dec (this was a hard deadline; labs are not submitted for late penalty pts after the deadline).

1. Assign 20% bonus calculated on the earned pts for a submission prior to 4:00am Mon 7 Dec.
2. Assign 10% bonus calculated on the earned pts for a submission between 4:00am Mon 7 Dec and 4:00am Tue 8 Dec.