Design Document:

By: Jordan Wakefield, Jacob Simons, and Rebekah Blazer

Overview:

To implement the three ciphers, we created a separate file for each cipher as well as a main file to be able to run each of the ciphers depending on what the user would like to do. Main first prompts for the cipher the user would like to use, then asks for the message and key they would like to use, then prints that to a new file as well as the terminal. When the user puts in invalid input, such as not using an alphabetical character or space or not choosing a valid mode, the user gets prompted until they enter valid input. By prompting users in this manner, they will have an easy time navigating the menu and prompts as well as clear messages informing them of what they need to do. After the user finishes with entering a message and key, the program exits with the output of their message and key in their appropriate files. We also have a unit test file to confirm the functionality of all ciphers.

How to run:

To run this program, in the terminal you are going to first need to compile the program. This was written in c++ so you will need to type “g++ CipherMain.cpp” (Assuming you have g++ installed on your computer). After the program is compiled, type “./a.out” or “a.exe” to run the program. From there, follow the prompts for encrypting or decrypting a message after choosing which cipher you would like to use. The same process can be applied to the Cipher_Unit_Tests.cpp. First type “g++ Cipher_Unit_Tests.cpp” and than run it via “./a.out” or “a.exe”.

Caesar Cipher:

In the Caesar Cipher file, there are three functions: caesar_encrypt(), caesar_decrypt(), and ceasar(). For caesar_encrypt(), it takes two strings as parameters, plaintext and shift, with the plaintext being the message to be encrypted and shift being the key to encrypt the message. Each character in the plaintext will be shifted based on the key that was given and the result is then returned. In caesar_decrypt(), we again take two strings as parameters, ciphertext and shift, where the ciphertext is the encrypted message and the key is what we shift by. This essentially reverts the message back to plaintext, and the result is returned. These parameters are required in order to encrypt or decrypt a message, by adding them as parameters they can be accessed and used in the algorithm to perform their operation. Finally, caesar() runs the prompts in main, asking for user input on what message and key they would like to encrypt or decrypt. The result is then output to a file as well as to the terminal to see results. Some error checking is included to make sure that valid input is being taken from the user, as well as outputting the appropriate message if the user gives invalid input.

An alternative was handling all keys as well as cipher implementations in main. This however would make main() look very bloated and difficult to read as things would not look very organized and be difficult to read. By creating a separate file for this implementation, the program looks very organized and is easier to read without having to spend a lot of time looking for minor details that may be causing errors.

Vigenere Cipher:

In the Vigenere Cipher file there are thee functions: vigenere(), vig_decryption(), and vig_encryption(). For vigenere() this takes in two parameters; string mode and string message_input. When this function is called, the inputed mode (which is a string that either says "ENCRYPT or DECRYPT) and the message have already been checked to make sure that that its uppercased alphabetical letters. The rest of the function gets a key from the user and makes sure that its alphabetical letters and that its less than the message. Finally it calls the correct encrypt and decrypt function and then outputs all relevant material (message, key, ciphertext, or plaintext) to the terminal and to the text files. For the vig_encryption()and vig_decryption() these functions take in a message and the newly gathered key from the previous function and do the proper vigenere cipher calculations and then returns the cipher/plaintext.

Since the requirements for the key are unique to the vigenere cipher alone, it made sense to handle this in this file. Throughout this file we treat A-Z as 0-25 with 26 being a SPACE.

I also include a testing file for Vigenere that just tests to make sure that not only is the function working, but also make sure that the outputted result is accurate and correct. These functions just print "passed" or "failed" to the terminal depending on the result of the test.

One Time Pad Cipher:

In the One Time Pad file there are three functions: otp(), Encrypt(), and Decrypt. The otp() function takes in two strings from CipherMain.cpp, string 'mode' and string 'message,' both imputed by the user. These parameters are taken from main as they are consistent with the other three ciphers and it saves coding space and time to have them set in main and taken into each individual cipher. When otp() is called, the user is prompted to enter a key from the terminal. Since the parameters for the key are different for each cipher, the handling of the key and error checking said key is done within the one_time_pad.cpp. The user can only enter letters and spaces, and the key must be the same length as the entered message, and will continuously be re-prompted until a valid key is chosen so that the user does not have to start over in main to retry their cipher. The valid key is then changed to uppercase and stored in the 'Key.txt.'

After the key has been stored and displayed the program checks to see if the mode selected was ‘ENCRYPT’ or ‘DECRYPT.’ If ‘ENCRYPT’ was chosen in main, then otp() will print the original message in all uppercase as well as the key in all uppercase before running Encrypt(). Encrypt() takes in string message and string key. Using a for loop, each letter of the message and of key are checked to see if they contain a space and encrypted. If they do contain a space, 59 is added to them to change their ascii value. This is done because the ascii value for a space is 32, and by adding 59, the value is changed to be right after ‘Z’ which is required for accurate encryption. The new character of ‘message’ is then calculated by adding the value of the ‘key’ character minus ‘A’ to the ‘message’ character minus ‘A’. ‘A’ is subtracted to change the values to 0-26 which is needed for proper encryption. That value is then modded by 27 to keep the cipher withing alphabetic bounds of A-Z and ‘A’ is added back to return the character to the proper ascii value. This new character is then also checked for ‘],’ the character after ‘Z,’ making it a space, and 59 is subsequently subtracted from it returning it to a space. This character is added to ‘Encrypt.txt’ and the full encrypted string returned to opt() where it is then printed to the terminal.

If ‘DECRYPT’ is chosen the process is very similar, the only difference being ‘message[i] and ‘key[i]’ are subtracted from each other and 27 is added after this subtraction. This is to correct negative values that occur during decryption. The resulting decrypted message is stored in ‘Decrypt.txt’ and returned to otp() where it is printed to the terminal.