Introduction

Since the early days of computing, computers have been used to help break encrypted messages. In this project, you will be “brute force” cracking codes encrypted with a *transposition cipher*, which is a method for scrambling messages without changing any of the actual characters in the message — in other words, the encrypted message is an anagram of the original message. Transposition ciphers were directly used by nations up until World War I to send messages, and they have been used as partial encryption methods in many encryption schemes since then.

The messages we will be using will be comprised of lowercase letters only: a, b, …, z. Mathematically, the difficulty level in decrypting an anagram increases by the factorial of the number of letters in the message (without including duplicates), and therefore it can be extremely difficult to decode a transposition cipher unless you know something else about the structure of the message. You will be writing a program that can decrypt a message using an English-language dictionary (although you could substitute any other dictionary for different languages), and you will be given a file with top passwords to use as keys. If you wish, you may change the encryption algorithm to use a different type of cipher, but that isn’t necessary for this project.

Definitions:

Word	Definition
encipher / encrypt	To change a readable message into a non-readable message via a key
decipher / decrypt	To change an encrypted message into a readable message via a key
key	A set of characters used to perform the encryption and decryption algorithm, e.g, “`abc123`”
brute force	Checking a set of keys via either a list of keys or a permutation of all possible keys
plaintext	The readable, unencrypted message, e.g., “`meet me at seven in halligan`”
ciphertext	The unreadable, encrypted message, e.g., “`ttnan___l_msil_meenie_v_geaeha`"

Your Program’s User Interface

Your program should work as follows:

For both Phase 0 and Phase 1:

To encrypt a phrase with a particular key:

```
./cipher --encrypt monkey
Please enter text to encrypt:
this is some secret text

Ciphertext:
_mrxsocets_tiseth_s_ieet
```

To decrypt a phrase with a particular key:

```
./cipher --decrypt monkey
Please enter text to decrypt:
_mrxsocets_tiseth_s_ieet

Plaintext:
this is some secret text
```

For Phase 0 only:

To decrypt a phrase with a dictionary and a key file:

```
./cipher --keyfile filename
Please enter text to decrypt:
_mrxsocets_tiseth_s_ieet

'ht sii smoses rcee txett' decrypted with key 'shadow'

'i  hstism  oseer sc etx  ett' decrypted with key 'letmein'

'this is some secret text' decrypted with key 'monkey'

'ctsex sieh esm st itorte' decrypted with key 'football'

'ihi sts emoseserc t txet' decrypted with key 'abc123'

'c eisxtsemes sh trteto i' decrypted with key 'baseball'

'si thioems scer seetxt t' decrypted with key 'dragon'

' o timctherei extsstsse  ' decrypted with key '12345'

'sh tiio mssecsr eee xttt' decrypted with key 'qwerty'

's tsecix mh eesstr itteo' decrypted with key 'password'
```

For Phase 1 only:

To decrypt a phrase with a dictionary (`words.txt`) and a key file:

```
./cipher --keyfile filename
Please enter text to decrypt:
_mrxsocets_tiseth_s_ieet

Checking: shadow, letmein, monkey, football, abc123, baseball, dragon, 12345,
qwerty, password,

Top 5 matches:
'this is some secret text' decrypted with key 'monkey': 100%
'ctsex sieh esm st itorte' decrypted with key 'football': 40%
'ihi sts emoseserc t txet' decrypted with key 'abc123': 40%
'sh tiio mssecsr eee xttt' decrypted with key 'qwerty': 40%
'ht sii smoses rcee txett' decrypted with key 'shadow': 20%
```

Getting the Files and Where to Start

Create a proj2 folder in your Desktop/comp11 folder. Then type:

```
/comp/11/files/proj2/getfiles
```

You will probably want to start by looking through the header files:

```
cipher.h      dictionary.h      keylist.h
```

Pay close attention to the public methods of cipher.h. That said, it is *very* easy to use the Cipher class, and there is an example of how to use it in both `keylist_tester.cpp` and in `phase0.cpp`.

If `keylist.h` is starting to look confusing, read the "Extra Information" below about the structs in the KeyList class.

Next, take a look at `keylist_tester.cpp`. This is a very simple program that will test your KeyList class, including the iterator (see below about iterators). The expected output for `keylist_tester.cpp` is in the `keylist_tester.output` file (which you can open in vim). You should compile and test `keylist_tester.cpp` (see how to compile it inside the file).

Much of `phase0.cpp` is written for you. You can probably fill in the `decrypt()` function almost immediately.

Once you have familiarized yourself with the project, start thinking about how to implement the KeyList class. It is a standard linked list, so use lab9 as an example, and also use the notes from class.

Required for Project

You have a lot of leeway for this project. But, you must write at least two classes: one is called the `KeyList` class, and the other is called the `Dictionary` class. The `KeyList` class will be a linked list implementation that will read a key file and then use those keys to brute force the decryption. The `Dictionary` class will be a dynamic array that will read in the dictionary file and will use be able to determine the percentage of words in a sentence that are in English.

We have provided you with header file for the `KeyList` and the `Dictionary` classes, which you should use. If you don't use those exact class definitions, you must still implement a linked list to hold the keys, and a dynamic array to hold the dictionary.

We have also provide you some template code for the `dictionary.cpp` and `keylist.cpp` files, including the code to read in the dictionary file and the list of keys. We have also provided you a small program called `keylist_tester.cpp` that you can use to test your `KeyList` class.

Finally, we have given you a template of the `phase0.cpp` file, which you can fill in and submit. It reads parameters directly from the command line, and that part of the assignment is built for you.

The encryption and decryption functions will be done through a Cipher class. The cipher class has a very simple interface, and allows you to define a Cipher object as follows:

```
Cipher cipher("encrypt", "key", "this is something to encrypt");

Cipher cipher("decrypt", "key", "h__mhgony_tssotntertiisei__cp_");
```

When you create a cipher in this method, it immediately encrypts or decrypts the message. You can retrieve the results of the encryption or decryption through the following three public methods:

```
                string get_plaintext();

                string get_ciphertext();

                string get_key();
```

Required: **`readme.txt`** file

All students will need to submit a `readme.txt` file with their provide submission. A `readme.txt` file spells out for the user a number of things (you should check all of the boxes below):

☐ **Your name, the date, the class (COMP 11), and the project (Project 2)**

☐ **The design, including a list of all of your classes and their purpose. E.g.,**

```
Dictionary Class
        This class is a dynamic array that reads in a dictionary
        from a file, and has the following public functions:
                Dictionary()  : the default constructor
                ~Dictionary() : the default destructor
                bool has_word(std::string word) : returns true
                         if the word is in the dictionary.

        The class has the following private methods:
                void expand() : to expand the dynamic array
                void read_dict_from_file(std::string filename) : this
                        reads in the words from a dictionary into the
                        dynamic array

        The class has the following private variables:
                std::string wordlist : this holds the dynamic
                                       array pointer
                int num_elements : the number of elements in the array
                int capacity     : the number of elements the array can hold
```

☐ **Operating Instructions: This tells the user how to use your program (e.g., "the program expects an input string in either plaintext or ciphertext, and ciphertext should only include lowercase letters and underscores. The command line arguments should be either "--encrypt key" or "--decrypt key" or "--decrypt key_length")**

☐ **Techincal details: A list of all of your files (just the names, including all the header files and Readme.txt), and instructions on how to compile the file, e.g. (you may have different files):**

```
To compile (all on one line):

  clang++ -Wall -Wextra -g cipher.cpp keylist.cpp phase0.cpp -o phase0
```

Phases and Submitting Your Work

Phase 2a: Due on Sunday, April 17, Midnight

For Phase A, you must have a completed KeyList class, and you must be able to print out the decryption of a string for all keys in the 10Passwords.txt file. You should also be able to encrypt and decrypt using a single key as shown in the User Interface section above. You should fill in the following checklist:

☐ **Able to print out an encryption and decryption for a single key**

☐ **Linked list class built with all functions implemented**

❏ **Logic for using the iterator in the KeyList class to walk through the entire list so you can decrypt the ciphertext using each key.**

❏ **Print all the decryption attempts to the screen, as shown in the User Interface above.**

❏ **Provide:**

```
      provide comp11 proj2a keylist.cpp keylist.h phase0.cpp readme.txt
```

Phase 2b: Due on Sunday, April 24, Midnight

For Phase B, you must have a working dynamic array Dictionary class that can read in a dictionary from the "words.txt" file. You must use this dictionary to determine what percentage of the words in a decrypted sentence are found in the dictionary. You must then print out the top five sentences, with the key used to decrypt them, and with the percentage of words in each sentence that are in the dictionary. See the User Interface section above for an example.

You should complete the following checklist for this phase:

❏ **Have a working Dictionary class with a dynamic array to hold the words**

❏ **Be able to search through the list to determine if a particular word is in the list**

❏ **Can determine the number of words in a sentence that are found in the dictionary**

❏ **Report on the top five decrypted sentences, based on the percentage of words correct.**

❏ **Provide:**

```
      provide comp11 proj2b dictionary.cpp dictionary.h
                            keylist.cpp keylist.h phase1.cpp readme.txt
```

Above and Beyond Ideas:

1AB. Implement a **binary search** to drastically increase the speed necessary to determine if a word is in the dictionary.

2AB. Using the key **length** alone, brute force a decryption based on *any* key which may or may not be in the dictionary. You can stop once you reach a sentence that has all English words (based on the same dictionary used above).

3AB. Replace the Transposition cipher with a different cipher that is harder to break.

Extra Information

Reading from the Command Line

When you type in a command to the linux terminal, you can type command line arguments as well as the name of the program. You do this all the time to compile, change a directory, and provide. In C++, you get access to your program's command line arguments by writing the main function like this:

```
int main(int argc, char *argv[])
```

The "`argc`" argument holds the number of arguments (including the name of your program as the first argument), and the "argv" argument holds strings for each argument in an array. You can access the second argument (the first argument after the program name) as follows:

```
string first_argument = argv[1];
```

We have set up the argument reading and parsing for you for this assignment.

The `KeyWithResult` struct and the `Node` struct

In the KeyList class, you will see two structs:

```
// KeyWithResult Struct
// Used to store the key for a decryption attempt
struct KeyWithResult {
        std::string key;        // the key for the attempt
        float word_percentage;  // the percentage of words that are in english
        std::string plaintext; // the resulting string after decryption
};

// Node for the list
// The data is a KeyWithResult struct
// The next is a pointer to the next Node
struct Node {
        KeyWithResult kwr;
        Node *next;
};
```

Your linked list will be similar to the linked lists we discussed in class, however the data will not be a simple type, it will be a `KeyWithResult` type called "`kwr`". The linked list mechanics are identical to the other linked lists we have covered before.

The `KeyWithResult` struct is simply this: it holds a `key` that you will read in from a key file, the `word_percentage` that you will get by comparing all the decrypted words in the dictionary, and the `ciphertext` you will receive from the Cipher class when you decrypt using the key.

See the `keylist_tester.cpp` file for an example of how you will use this struct.

Iterators

In the KeyList class, you will see an extra variable we haven't seen before in a linked list:

```
            Node *iterator;
```

An "iterator" is simply a helpful tool to allow you to traverse through a list one element at a time. The iterator pointer is changed in three places: whenever you `insert()` into the list, the iterator gets reset to the head of the list; whenever you call the `reset_iterator()` function, the iterator gets reset to the head, and whenever you call the `next()` function, the iterator is set to the next element in the list. The `next()` function returns the data (a `KeyWithResult` struct) from the current iterator Node, and then updates iterator to point to the next node.

The Transposition Cipher

It is not necessary to understand the material below for this assignment.

A "transposition cipher" is a method for encrypting a message by transposing the characters in the message by putting the message into a rectangular grid and reading down the columns in an order based on a textual key.

For example, the message “`meet me at seven in halligan`" (called the *plaintext*) with the key "`COMPSCI`" would be put into a grid like this (with underscores representing spaces):

```
C O M P S C I
-------------
m e e t _ m e
_ a t _ s e v
e n _ i n _ h
a l l i g a n
```

The letters would then be read out in columns, based on the alphabetic ordering of the letters in the key:

```
C O M P S C I
0 4 3 5 6 1 2
```

The first column that would be read would be `M_EA`, then `ME_A`, then `EVHN`, etc., for a final encryption of:

```
m_ea me_a evhn et_l eanl t_ii _sng
```

The final output is called the *ciphertext*, and that is what you can pass to your friends on the other side of enemy lines.

To decrypt an encrypted message, you need to know the key, and you can reverse the process to get back the original message. First, put the message into columns based on the length of the key:

```
C C I M O P S
0 1 2 3 4 5 6
-------------
m m e e e t _
_ e v t a _ s
e _ h _ n i n
a a n l l i g
```

Going back to the alphabetic ordering of the key, and putting the current columns under them, this tells us how to re-order the current columns:

```
C O M P S C I
0 4 3 5 6 1 2 (key alphabetic order)
0 1 2 3 4 5 6 (ciphertext column order)
```

The translation from the ciphertext grid columns to the actual output columns is as follows:

```
ciphertext column      plaintext column
0                   ->        0
4                   ->        1
3                   ->        2
```

```
5                    ->         3
6                    ->         4
1                    ->         5
2                    ->         6
```

In other words, we rearrange the columns like this:

```
0 4 3 5 6 1 2
-------------
m e e t _ m e
_ a t _ s e v
e n _ i n _ h
a l l i g a n
```

Finally, the original plaintext is read back by reading across the grid by rows, to produce:

```
meet_me_at_seven_in_halligan
```