CSE232 Fall 2018

Programming Project #5

Assignment Overview

In this assignment we will create a program that supports the encryption and decryption of text. This assignment is worth 40 points (4.0% of the course grade) and must be completed and turned in before 11:59pm on Monday, October 15th . That's two weeks because of the upcoming midterm on Thur, October 4th.

Description

Most cyphers, like a Caesar cypher or a Rot13 cypher, are substitution cyphers. In a substitution cypher one substitutes one letter for another using some simple algorithm. For example, here is a Rot13 cypher (a right rotation of the alphabet by 13).

Original: `a b c e f g h i j k l m n o p q r s t u v w x y z`
Encoding: `n o p q r s t u v w x y z a b c e f g h i j k l m`

To generate a secret message, you substitute the original letter with its associated encoding:

```
hello ❼ tqxxa
```

To decode, you just do the reverse. Simple eh? Yes, and easy to break. Substitution cyphers are very vulnerable to statistical measures. We know the frequency of letter usage in English (in any language) and, if we have enough text, we can figure out the encoding using those statistics. See https://en.wikipedia.org/wiki/Letter_frequency

How to get around this vulnerability? Well, there are many approaches but one, relatively simple, approach is to encode more than just single letters. What if we encode every **pair** of letters and substitute one **pair** for another? This is a digraph cypher, so called because pairs of letters are called digrams (also called bigrams). It was first used in the 1850's and called the Playfair cypher https://en.wikipedia.org/wiki/Playfair_cipher

Why would this be any better? Well, now you have a much bigger set that you have to break. Instead of 26 substitutions, you have 26x26 substitutions (676) to find. Though digram/bigram frequency is available, the process is more difficult. We could do trigrams, quadgram, quintgrams even! http://practicalcryptography.com/cryptanalysis/letter-frequencies-various-languages/english-letter-frequencies/ but we'll stick with digrams.

The Playfair Algorithm

Plaintext Preparation

We prepare the plaintext, the string we are going to encode, as follows.

- we will only represent the characters a-z (minus 'j', which is not represented). Any capitalized letters are converted to lower case. Any other character is removed from the plaintext. Thus numbers, spaces, punctuation and the letter 'j' and 'J' are removed.

- if any double letters occur such as "hammer", "upper" we replace the second letter with the letter 'x', making "hamxer", "upxer".
- if the length of the plaintext is odd, we make it even by append 'x'

Key square Preparation

We create a key square by first requesting a key word. Let's use the keyword "desperate". We fill in a 5x5 square such that it begins with the keyword, but does not allow the repetition of any letter. Once finished the remaining letters in a-z (minus j) are filled into the square *alphabetically*.

```
d e s p r
a t b c f
g h i k l
m n o q u
v w x y z
```

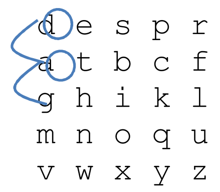
Notice that the 'e' in "desperate" is not repeated. Only one example of each letter is allowed. The rest of the letters are those that did not already occur in alphabetic order.

As a simple string, the key square would be `"despratbcfghiklmnoquvwxyz".`
While it will be easier to view the encoding/decoding process in 2D, it is not necessary to use a 2D vector in the algorithm as we will see.

Encoding

We take the pairs of letters from the plaintext and substitute them with a different pair using the key square. There are three general cases. Let's encode the message "danger"

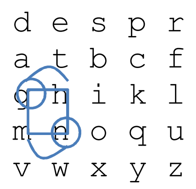
First pair: "da", letters are in the same column
For each letter, find the letter just below. If the letter is at the bottom of a column, use column wrap and use the letter at the top of the same column.



```
d e s p r
a t b c f
g h i k l
m n o q u
v w x y z
```

da ❼ ag

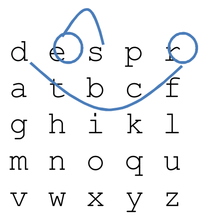
Second pair: "ng", letters are in different rows and columns.
As such, the two letters form two corners of a square. We use the other two corners as the letter we substitute. The order is important – the first encrypted letter of the pair is the one that lies on the same row as the first plaintext letter.



```
d e s p r
a t b c f
g h i k l
m n o q u
v w x y z
```

ng ❼ mh

Third pair: "er". Both letters are on the same row.
Like the same column pair, we use the next letter shifted down from each of the original letters. If the letter in question is at the end of the row, we wrap around to the beginning of the row.



```
d e s p r
a t b c f
g h i k l
m n o q u
v w x y z
```

er ❼ sd

So using the keyword "desperate", we can encode "danger" as "agmhsd"

Decoding
Same process, just undo the encoding.

Representing a 2D block as a string
It seems like you would need a 2D vector or array to do what we just described, but you really don't. You have to realize the following relationship between a 1D string and a 2D block. Let's work with our previous key square. Look at the relationship between the 1D string "`despratbcfghiklmnoquvwxyz`" and that 2D block. Row and Column numbers are added for clarity for the 2D Block, indices added to the 1D string.

```
  0 1 2 3 4
0 d e s p r
1 a t b c f
2 g h i k l
3 m n o q u
4 v w x y z
```

```
          1111111111222222
01234567890123456789012345
despratbcfghiklmnoquvwxyz
```

Let's look at the letter 'h'. The 'h' in this string occurs at index 11 (remember, first index is 0) in the 1D string, or at row 2, column 1 of the 2D block. In the block, each row occupies 5 elements, and there are 5 rows. The relationship between the 1D index and the 2D row/col values are:

row = index / 5 (using integer division).	11 / 5 → 2	/ 5 because there are 5 rows
col = index % 5 (modulus)	11 % 5 → 1	% 5 because 5 elements/row
index = row * 5 + col		2 * 5 + 1 → 11

Thus, we can find the equivalent 2D row/col values if we have a 1D index in a string, and if we have the 2D row/col values we can convert that to a 1D index in a string. That's all we need.

Program specifications

Note that we pass all strings as const references. We pass them as references to save work (no copy of arguments required). We make them const so they cannot be modified.

function `string prepare_plaintext(const string &s)`

- one argument, a string
- return, a string

Takes the string, makes all alphabets lowercase, strips/removes any non a-i, k-z characters, replaces the second letter of a double letter with 'x', makes the length of the string even (adding 'x' to the end if necessary) and returns that string.

function `string create_encoding(const string &key)`

- one argument, a string (the keyword)
- return, a string (the key square)

Creates a key square, as a 1D string, as described above.

function
```
string encode_pair(const string &pr,
                   const string &key)
```

- two arguments
    - a string of two characters to be encoded
    - a string key square
- return, the new encoded pair string

Encode a pair from the plaintext

function
```
string encode(const string &plaintxt,
              const string &key)
```

- two arguments
    - the string plaintext
    - the string key square
- return a string, the encoded plaintext pair

Runs the provided plaintext through `prepare_plaintext`. Encodes all the pairs in the plaintext. Uses `encode_pair`

function
```
string decode_pair(const string &pr,
                   const string &key)
```

- two arguments
    - a string of two characters to be decoded

    - a string key square
- return, the decoded plaintext

Decode a pair from the plaintext

function `string decode(const string &encodedtxt,`
`                      const string &key)`

- two arguments
    - the string encoded text
    - the string key square
- return a string, the decoded encrypted text

Decodes all the pairs in the plaintext. Uses `decode_pair`

Deliverables

You will turn in one file: `proj05_functions.cpp`. We provide you only with `proj05_functions_05.h` , you must write your own main to test your functions. Mimir can test the individual functions without a main program but it's a good idea for **you** to test your own code with a main, perhaps in the manner that we did previously.

Remember to include your section, the date, project number and comments and you ***do not provide*** main.cpp. If you turn in a main with your code Mimir will not be able to grade you.

1. Please be sure to use the specified file names
2. Always a good idea to save a copy of your file in your H: drive on EGR.
3. Submit to Mimir as always. There will be a mix of visible and not-visible cases.

Assignment Notes

1. You turn in the functions only. To test against your own main you can write a separate file with the main and then compile the two files at the same time. See the lab and videos for examples.