COMP161 - Project 1 - Simple Substitution Ciphers

Spring 2015

Your first project is a simple CLI-based encryption and decryption program.

Monoalphabetic Substitution Ciphers

Computer science has a long and rich history with cryptography. For this project you'll implement a simple cipher from the days of pre-computing cryptography: a monoalphabetic substitution cipher. Let's begin with the classic example of a monoalphebetic substitution system: the Caesar shift.

First we choose a KEY by shifting our 26 letter alphabet. For this example we'll shift the letters to the right by three. This produces the following sequence of letters:

```
XYZABCDEFGHIJKLMNOPQRSTUVW
```

When set below the standard alphabet[1], our shifted sequence defines a substitution pattern that we can use to obscure written text. The standard alphabet represents letters in our original text, called the PLAINTEXT. The shifted alphabet represents their counterparts in our ENCODED text, called the CIPHERTEXT[2].

[1] A to Z in order

[2] we call the alphabets the plain and cipher alphabets respectively

```
plain:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
cipher: XYZABCDEFGHIJKLMNOPQRSTUVW
```

Let's say we wished to encode the message "My secret" using our cipher. Then all we need to do is substitute every letter in the message with it's cipher equivalent. For now we ignore letter case, spacing, and punctuation. Using our key, "My secret" becomes "Jv pbzobq"[3]. To reverse the process, or DECODE or ciphertext, we simply read our key backwards[4].

In general, any reordering[5] of the standard alphabet defines a simple substitution alphabet. These systems are called monoalphabetic substitution ciphers because this single alphabet defines the complete substitution pattern used for encoding and decoding messages. The Caesar shift is the simplest, most well known, and least secure means of generating a single substitution alphabet. In general, monoalphabetic systems have a long history but provide little security in the modern world.

[3] substitute M with J, y with v, s with p, and so forth

[4] Substitute J with M, v with y, p with s, etc.

[5] permutation

Leaving in things like upper and lower case letters, word breaks, and punctuation makes guessing the key much easier. In fact, people solve these types of puzzles for fun all the time. If we want to get the most security possible out of our monoalphabetic substitution system, then we can do the following transformations to our plaintext before encoding.

1. Remove punctuation
2. Remove white space and word breaks.
3. Group letters in groups of 5
4. Pad the message so that its length is a multiple of 5

The five letter groups are a good size for reducing human encoding and decoding error. They don't really provide extra security. We'll keep with tradition even though we're using a computer based system. If we really want to squeeze more security out of our system, then we pad the message with letters that occur infrequently in our message or in English generally[6]. If we return to "My secret" and apply these transformations we end up with something like "MYSEC RETJJ"[7] which then is then encoded to "JVPBZ OBQGG".

[6] The frequency of letters in English is the main way we attack the decipherment of these messages without knowing the key

[7] Padded with J

Caesar shift keys are extremely weak because you can quickly and easily check all possible shifts and find the decipherment that looks like English. They are great for humans because you really just have to remember the shift amount and not the whole sequence. In class we'll look at a few fun ways to come up with human-friendly keys. The best keys are the least human friendly: random permutations of the the alphabet. Thankfully we have computers, so we won't really need to worry about remembering a random reordering[8] of the alphabet.

[8] permutation

The Program Requirements

For your project, you'll be writing a CLI-based program that allows users to do the following:

1. Encode a plaintext message given a key. Both the plaintext and key are read from files. The resultant ciphertext is written either to a file or the standard output. Prior to encoding, the plaintext message should be preprocessed as described above.
2. Decode a ciphertext message given a key. Both the ciphertext and key are read from files. The resultant plaintext is written either to a file or to the standard output.
3. Generate a random key and write it either to a file or the standard output.

Your program will have a simple, linux-like interface. For the following examples, assume the program executable is named *monoalpha*. To generate a random key users do the following:

```
monoalpha -k [KEYFILE]
```

When given the optional filename *KEYFILE*, the program writes a random permutation of the standard alphabet to the file. If it is omitted, then the key is written to the standard output. Here's some examples of what this might look like in practice[9].

[9] $> is the prompt

```
$> monoalpha -k
XYZABCDEFGHIJKLMNOPQRSTUVW
$> monoalpha -k key.txt
$>
```

In the second example the file *key.txt* would either be created and filled with the key or its contents would be overwritten with a key. The key will be written to a file the same way it is written to the standard output: a single line of all uppercase letters.

To encode a plaintext document we'll require the key and the plaintext to be in files. The ciphertext will optionally be written to a file or the standard out in a fashion similar to that of the key generation command.

```
monoalpha -e keyfile plaintext [CIPHERTEXT]
```

We'll place no requirements on the formatting of plaintext documents, but our encoder will preprocess the text as previously discussed. The resultant ciphertext will, therefore, be written in blocks of five uppercase letters. The *keyfile* and *plaintext* arguments should be the file names for the key and plaintext respectively. The file containing the key should be a file like those written by the key generating routine of our program.

Decoding ciphertext is similar to encoding plaintext. The ciphertext and key must be in files and the plaintext is optionally written to the standard output or a file.

```
monoalpha -d keyfile ciphertext [PLAINTEXT]
```

We'll require that ciphertext documents be written in blocks of all uppercase letters where each block contains five letters. Put another way, the ciphertext should be a possible output of our program's encoding routine. The plaintext will be written in blocks of five uppercase letters. Our program does not attempt to reconstruct the words of our original message. It simply substitutes ciphertext letters for plaintext letters. The file containing the key should be a file like those written by the key generating routine of our program.

Finally, we should provide some help text to the user.

```
monoalpha -h
```

The same text can be used in the event that the user passes poorly formatted CLI arguments to our program.

Logistics

You'll have just over two weeks to complete this project. All the lab periods during that time are dedicated to the project. The first lab period has a specific assignment attached to it where the second is time to work and get assistance. You should be putting some serious time into this project prior to that second lab so that you may spend time getting help on tricky parts and not spending time setting up files and getting started.

Project Lab

At the core of this project are three processes: preprocessing plaintext, encoding processed plaintext, and decoding ciphertext. In lab you'll work through each step the old fashioned way: by hand. **This work must be checked prior to the end of the lab period.**

1. Rewrite the following message in the "blocks of five" style described above.

   > All's fair in love, war, and crypto.

   Now develop a key and encode the resultant message.

2. Use the key *ONMLKJIHGFEDCBAZYWXVUTSRQP* to decode this message:

   > DKVCK VKDDQ AUOXK MWKVP

It makes a lot of sense to envision these processes as C++ procedures working with std::string data and then build up an interface around them to complete the project. For the coding portion of this lab you start do just that:

1. Produce a library header with proper declarations and documentation for each of the three core procedures.
2. Put stubs for these procedures in a corresponding library implementation.
3. Re-write the examples you did by hand as C++ unit tests for these procedures[10]

[10] note that that your final project submissions may require further tests, so you may include extra tests in your submission if you wish.

Submit your code as assignment *labp1* using *handin*.

Due Dates

Assignment	Due Date
Project Lab (Handwritten)	End of lab on Wednesday
Project Lab (Code)	Class time on Friday 3/27
Project	Class time on Wednesday 4/8

Grading

Your project will be evaluated on the following criteria:

- Completeness,correctness and evidence of an iterative development process

  Ideally, your program is a complete working version of the program described above. To get to that point you should be working through a series of iterations, where each iteration adds some functionality to the program and ensures the correctness of that functionality inappropriate ways. *This means that incomplete programs should clearly be a finished version of something and not an unfinished program.* This also means that the finished product *should always compile, run, and do something.*

- Usage of good program/procedure design principles

  We have a standard process for designing, documenting, and developing procedures. Lack of documentation and proper unit tests will result in a loss of points regardless of the level of completion or apparent correctness of the program itself.

- Usage of good code style

  Ugly code that works is still ugly code. Human beings read more C++ than compilers do and human beings struggle when reading ugly code. Don't write ugly code. Properly indent your code[11] and follow our standard conventions for writing C++. You should also ensure that your code will not line wrap when printed[12].

[11] Emacs will do this for you if you let it

[12] Emacs will help you with this if you let it