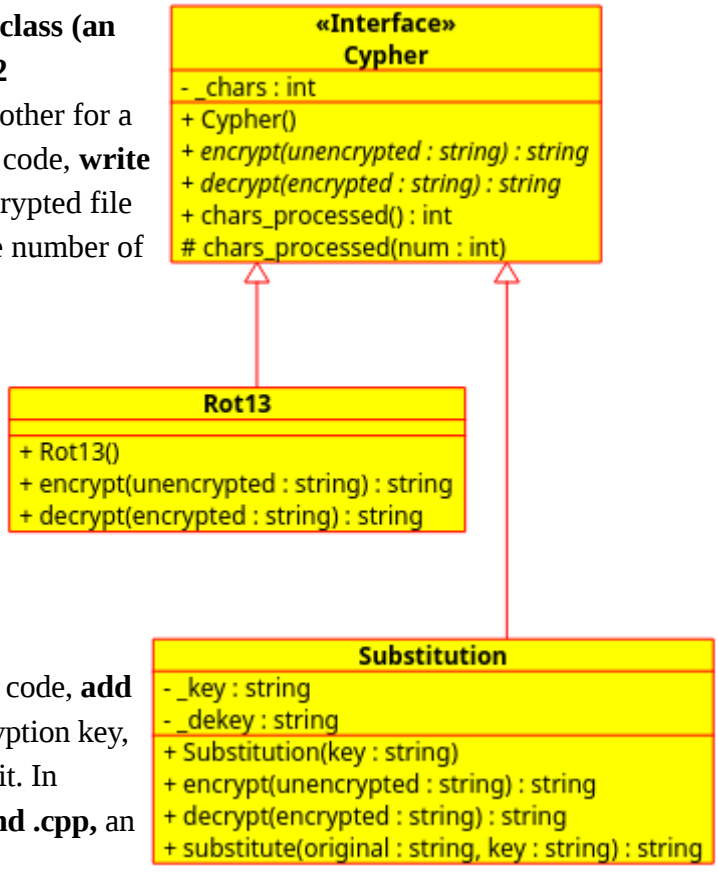
# Keeping Your Options Open – Cypher

CSE 1325 – Spring 2018 – Homework #4 **Update 2**[1]
Due Thursday, February 15 at 8:00 am

## Assignment Overview

Cryptography is the science of replacing information such as text with a different set of text, such that the original text can no longer be easily determined. Numerous algorithms exist for this, and you'll get to implement a few in this assignment. In the process, you'll practice file I/O as well as inheritance.

**Full Credit:** Using git (3+ commits), **define a virtual class (an *interface*) for encrypting and decrypting text**. **Derive 2 implementations** from this interface, one for Rot13, the other for a simple Substitution cypher. In addition to your usual test code, **write a main program** that will encrypt a file test.txt to an encrypted file test.txt.rot13 or test.txt.subst, or vice-versa, then print the number of characters encrypted or decrypted.




**Deliver file CSE1325_04.zip** to Blackboard. In **subdirectory "full_credit"**, include your full git repository, including files **main.cpp, cypher.h and .cpp, rot13.h** and **.cpp, substitution.h** and **.cpp**, **Makefile,** and **cypher1.png** through **n** (as many screenshots as you deem useful) demonstrating your tests and your main executing.

**Bonus:** Using git (3+ additional commits) and the above code, **add an XOR implementation** using an image file as the encryption key, producing text.txt.xor. **Update tests and main** to support it. In CSE1325_04.zip **subdirectory "bonus"**, include **xor.h and .cpp,** an **updated class diagram,** along with the above files.

**Extreme Bonus:** Using git (3+ additional commits), add a new class Async that implements any simply asynchronous key encryption algorithm (the common "best practice" actually used for Internet, credit card, and other serious security applications). **Update tests and main** to support it. In CSE1325_04.zip **subdirectory "extreme_bonus"**, include all files required for Bonus above as well as **async.h**, **async.cpp,** an **updated class diagram,** and any other supporting files you require. You have a lot of leeway in the design and implementation of this code!

---

1 Clarifies that the number of characters encrypted or decrypted should be printed at the end of main().

# Full Credit Requirements

In C++, the interface to a class is usually implemented in a header file. One way to provide different implementations of the same interface is for the header file for each implementation to inherit (become a derived class of) the interface class, including the interface's header file.

We'll try this out by **defining a Cypher interface in a header file**, then **implementing several cypher algorithms that each derive from Cypher**. Note that the **encrypt and decrypt methods are pure virtual** (italicized in the class diagram above).

In addition, **the Cypher class includes a static (class level) variable chars**, which keeps track of the number of characters encrypted or decrypted *by every instance of any class derived from Cypher*. This value should be printed at the end of main(). To make this work, the encrypt and decrypt methods implemented by Cypher's derived classes must call the **protected method chars_processed**, passing the number of characters encrypted or decrypted by that call. The value of chars is retrieved by calling a **public method chars_processed** that accepts no parameters.

Encryption is commonly used to hide or protect sensitive information, among other use cases. A very common example in the pre-web days of the Internet, when Usenet Newsgroups (something like Twitter without hashtags) was all the rage, was the Rot13 algorithm.

## Rot13

Rot13 stands for "Rotate 13" - it simply **adds 13 to the ASCII value of each character in a string, subtracting 26 if the resulting character is greater than 'z'**.

So "The quick brown fox jumps over the lazy dog" becomes "Gur dhvpx oebja sbk whzcf bire gur ynml qbt".

Newsgroup readers implemented Rot13 because of one unusual feature of Rot13: Encrypt again, and you get the original string. So encrypting "Gur dhvpx oebja sbk whzcf bire gur ynml qbt" with Rot13 a second time restores it to "The quick brown fox jumps over the lazy dog". This made Rot13 particularly useful for hiding spoilers, punch lines, and puzzle solutions.

## Substitution

Substitution cyphers have been around at least 2,500 years, and are popular among school children to this day because of their simplicity. The idea is to match each letter of the alphabet to a different, unique letter:

```
abcdefghijklmnopqrstuvwxyz
bfdhmojekixnwqrcltpsauzyvg
```

Then simply **substitute the lower letter for the upper letter for each letter in the message**. So "The quick brown fox jumps over the lazy dog" becomes "Sem lakdx ftrzq ory iawcp rumt sem nbgv hrj".

Unlike with Rot13, decryption requires the same key. Substitute the upper letter for the lower letter for each letter in the encrypted message. An easy way to do this is to create a decryption key, by placing the lower string in alphabetic order with the upper string staying in sync.

## User Interface

**The user interface is completely up to you.**

- You could prepare a **menu-driven program**, allowing the user to select whether to encrypt or decrypt the file (and perhaps a phrase as well), collect the filename, the cypher, and (for substitution) the key, and then process the file.
- You could define a **command-line interface**, with a parameter specifying the cypher (-c rot13 and -c substitution), -k the flag specifying the key, and the parameter a filename.
- You can use **any other interface** you believe would be efficient.

**The output file MUST be the same name as the input file, but with .rot13 or .subst appended** to the original filename when encrypting and removed when decrypting.

As with previous homework, the student will deliver a ZIP archive file named **CSE1325_04.zip** with full_credit, bonus, and extreme_bonus subdirectories.

The **full_credit** subdirectory will contain the local git repository (3+ commits) as usual, along with the following additional files :

- The source code files **main.cpp, cypher.h** and **cpp, rot13.h** and **cpp, substitution.h** and **.cpp,** and **test.cpp,**
- A correct **Makefile**, and
- Any screenshots you choose to include to assist the graders in giving you the best grade possible.

# Bonus Requirements

Another simple encryption approach relies on the exclusive or binary operator, ^. An exclusive or operation flips the original bit only if the associated key bit is 1, leaving it unchanged if the bit is 0. So given char ‘a’ (ASCII 0x61) and key ‘3’ (ASCII 0x33), ‘a’^’3’ is ‘R‘ (ASCII 0x52).

**0b0110 0001**
**0b0011 0011**
**0b0101 0010**

For the bonus, you’ll create a third derived class named Xor implementing the exclusive or cypher. In addition to the filename to encrypt, **also accept an independent filename to use as the key**. For each byte in the file to encrypt, read a corresponding byte from the key file. Write the exclusive or of the two bytes to the encrypted file, whose **filename will have .xor appended** on encryption and removed on decryption.

The exclusive or algorithm is like Rot13 – you can decrypt by using the same encryption process with the same key file.

For credit, the bonus subdirectory will contain the following source code files corresponding to the class diagram:

- The source code files **xor.h** and **.cpp**,
- **main.cpp** updated to also support the exclusive or cypher,
- An **updated class diagram** (.xmi or .uml file, with PNG screenshots recommended), and
- The same files as from the full credit assignment.

Hint: If you encrypt all characters in a file *except newlines*, keeping the original line structure intact, the encrypted text will contain *additional* newlines as a result of the exclusive or operations. One way to address this by ensuring that the newlines in the original file are also encrypted, and newlines in the encrypted file are also decrypted (think binary mode). You may implement a different solution to this issue as well, as long as your encryption and decryption process works.

# Extreme Bonus Requirements

Asynchronous key encryption, also known as public key encryption, relies on two mathematically related keys, one called “public” and the other “private”. A file encrypted with the public key cannot be decrypted with the public key, but rather only with the private key. Similarly, a file encrypted with the private key can only be decrypted with the public key.

This offers numerous advantages over the cryptographic algorithms implemented earlier.

- The public key may be freely published to the general public, while the private key need never leave the possession of the originator. This solves the problem of securely transmitting a secret key to the other party.
- A message encrypted with the recipient’s public key (freely and publicly available) can only be decrypted by that recipient, as only the recipient holds the private key.
- Similarly, a message encrypted by the sender’s private key has been digitally signed, as the fact that the sender’s public key decrypts the message is proof that the sender encrypted it with their private key.
- Encrypting a message with the recipient’s public key and then the sender’s private key sends a secret message that can be proven to have originated with the sender.

For extreme bonus credit, research and then implement any asynchronous key encryption algorithm as class Async derived from the Cypher interface above. Update your regression tests and main to support it and verify that it works properly.

For credit, the extreme_bonus subdirectory will contain the following:

- The source code files from the bonus directory updated to support the Async class, along with **async.h** and **.cpp,** an **updated class diagram**, and **any other files necessary to demonstrate your work.**