UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS110 Assignment 2

Modular Cryptography

Due date: 30 September 2016, 12h00 (midday)

Total marks: **60**

1 General instructions

- This assignment should be completed **individually**.
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- If your code does not compile you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence of certain functions or classes).
- Read the entire assignment thoroughly before you start coding.
- To ensure that you did not plagiarize, your code will be inspected with the help of dedicated software.
- Note that **plagiarism** is considered a very serious offence. Plagiarism will not be tolerated and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `http://www.ais.up.ac.za/plagiarism/index.htm`.

2 Overview

Cryptography is a method of storing and transmitting data in a particular form so that only those for whom it is intended can read and process it. In today's computer-centric world, cryptography is most often associated with scrambling plaintext (ordinary text, sometimes referred to as cleartext) into ciphertext (a process called encryption), and then "unscrambling" it back again (decryption). For this assignment, you will create a hierarchy of classic encryption algorithms, and a pipeline method to combine various algorithms together for stronger encryption.

3 Your task

This assignment will give you an oppurtunity to work with inheritance, polymorphism, aggregation, STL vectors, exception handling, and operator overloading. You will create 8 classes that interact through inheritance and aggregation. Create the classes as they are discussed for each task below. Test your work for each task thoroughly before submitting to the corresponding fitchfork upload slot and moving on to the next task. Each task is evaluated separately, but will rely on the classes you created in previous tasks.

The classes you will create for each task are:

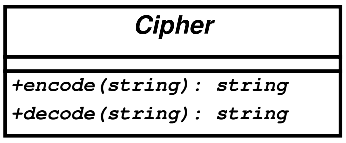
1. `Cipher, SubstitutionCipher, Caesar, OneTimePad`
2. `TranspositionCipher, RowsColumns, ZigZag`
3. `CipherPipeline`

3.1 Task 1: Substitution cipher hierarchy (20 marks)

There are two major types of classic text ciphers: substitution ciphers and transposition ciphers. In substitution ciphers, each letter of plaintext is encoded and decoded individually. For this task, you will create a simple hierarchy of classes leading to two concrete substitution ciphers: Caesar cipher and One Time Pad cipher.

3.1.1 Cipher

The `Cipher` class is an abstract class that describes the basic properties of any cipher. This is an interface class, i.e. it provides no implementation. Create `Cipher` class in a file called `Cipher.h` according to the specifications below:



A cipher is an algorithm that can encode plaintext, turning it into ciphertext, as well as decode ciphertext, turning it back into plaintext. Abstract Cipher declares two pure virtual functions:

- `string encode(string)` – This function is abstract (i.e. pure virtual) and must be overridden by the derived classes. The function will receive plaintext (`string`) as input, encrypt it, and return encrypted ciphertext (`string`)
- `string decode(string)` – This function is abstract (i.e. pure virtual) and must be overridden by the derived classes. The function will receive ciphertext (`string`) as input, decode it, and return decoded plaintext (`string`).

3.1.2 SubstitutionCipher

In cryptography, a substitution cipher is a method of encoding by which individual units of plaintext are replaced with ciphertext, according to a predefined algorithm. For this assignment, "units" of text are single characters. Substitution ciphers will thus encode and decode text by encoding/decoding each character of text individually.

The generic `Cipher` interface can be extended to create a generic `SubstitutionCipher` class. Create the `SubstitutionCipher` class in `SubstitutionCipher.h` and `SubstitutionCipher.cpp` files, according to the specifications below.

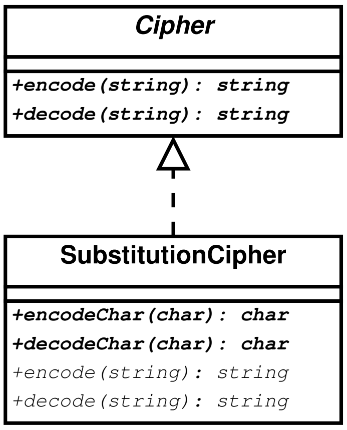
Since characters are going to be encrypted and decrypted individually, two functions can be added to the `SubstitutionCipher` class:

- `char encodeChar(char)` – This function is abstract (i.e. pure virtual) and must be overridden by the derived classes. The function will receive a plaintext character (`char`) as input, encrypt it, and return ciphertext character (`char`)
- `char decodeChar(char)` – This function is abstract (i.e. pure virtual) and must be overridden by the derived classes. The function will receive a ciphertext character (`char`) as input, decode it, and return plaintext character (`char`).

The abstract functions inherited from the parent class `Cipher` can be overidden and implemented now:

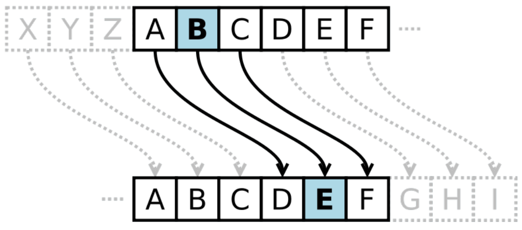
- `string encode(string)` – This function receives plaintext (`string`) as input, encrypts it by applying `encodeChar` function to each character of plaintext, and returns the ciphertext (`string`).
- `string decode(string)` – This function receives ciphertext (`string`) as input, decodes it by applying `decodeChar` to every character of ciphertext, and returns plaintext (`string`).

The following UML illustrates the relationship between `Cipher` and `SubstitutionCipher`:



3.1.3 Caesar

The Caesar cipher, also known as a shift cipher, is one of the simplest forms of encryption. It is a substitution cipher where each letter in the plaintext is replaced with a letter corresponding to a certain number of letters up or down in the alphabet. For example, with a right shift of 3, 'A' would be replaced by 'D', 'B' would become 'E', and so on, as illustrated below. The method is named after Julius Caesar, who used it in his private correspondence.



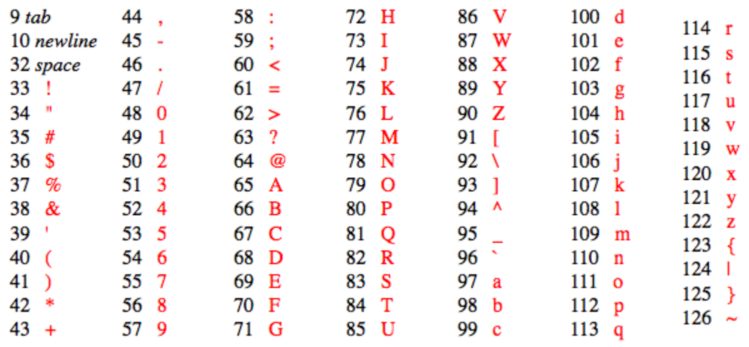
The transformation can be represented by aligning two alphabets; the cipher alphabet is the plain alphabet rotated left or right by some number of positions. For right shift of 3:

`ABCDEFGHIJKLMNOPQRSTUVWXYZ` – plain alphabet
`DEFGHIJKLMNOPQRSTUVWXYZABC` – cipher alphabet

When encrypting, each letter of the message in the "plain" line is matched with the corresponding letter in the "cipher" line, and the cipher letter is used in place of the plain letter. For example:

`HELLO WORLD` – plaintext
`KHOOR ZRUOG` – ciphertext

To decode, the process is reversed: ciphertext letters are replaced by the corresponding plaintext letters. In practice, the two alphabets do not have to be stored and aligned. Instead, we can operate on the ASCII values of the characters directly to perform the shift. The table below lists the ASCII codes of all printable characters:



9	*tab*	44	,	58	:	72	H	86	V	100	d	114	r
10	*newline*	45	-	59	;	73	I	87	W	101	e	115	s
32	*space*	46	.	60	<	74	J	88	X	102	f	116	t
33	!	47	/	61	=	75	K	89	Y	103	g	117	u
34	"	48	0	62	>	76	L	90	Z	104	h	118	v
35	#	49	1	63	?	77	M	91	[	105	i	119	w
36	$	50	2	64	@	78	N	92	\	106	j	120	x
37	%	51	3	65	A	79	O	93	]	107	k	121	y
38	&	52	4	66	B	80	P	94	^	108	l	122	z
39	'	53	5	67	C	81	Q	95	_	109	m	123	{
40	(	54	6	68	D	82	R	96	`	110	n	124	|
41	)	55	7	69	E	83	S	97	a	111	o	125	}
42	*	56	8	70	F	84	T	98	b	112	p	126	~
43	+	57	9	71	G	85	U	99	c	113	q		

Tabs and newlines (ASCII codes 9 and 10) add too much of a visual distortion and are not convenient to work with, therefore for this assignment we will limit the available range of codes to [32,126]. Thus, the available alphabet is made of 126 - 32 + 1 = 95 symbols. To further simplify the implementation, only right shifts will be allowed, and the maximum shift is 94.

To encode a character, all you need to do is add the predefined **shift** value to its ASCII code. Eg, to encode 'A' with shift = 3, add 3 to the ASCII value of A: 65 + 3 = 68. 68 is the ASCII code of 'D'.

To decode a character, the same shift has to be subtracted: Decoding 'D' implies subtracting the shift, 68 - 3 = 65, giving you 'A'.

What will happen if you try to shift '}' (ASCII code: 125)? 125 + 3 = 128, which is out of range. In this case, encoding has to be "wrapped around" the ASCII table: once you reach the end, go back to the beginning (code 32). Thus, shifting '}' by 3 positions should take you to '!'. Same applies to decoding: if subtracting a shift value produces an out-of-printable-range code, wrap it around the table. Decoding '!' should give you '}'.

The `Caesar` class will have a protected member `shift`:

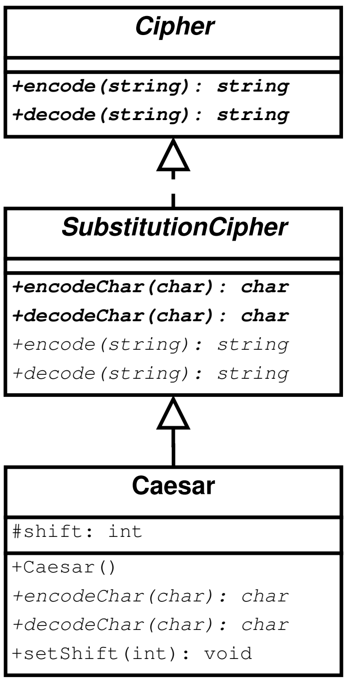
- `int shift` – The shift value used for encoding/decoding as described above.

The `Caesar` class will have the following member functions:

- `Caesar()` – Default constructor that takes no parameters. Shift value is set to 0.
- `void setShift(int)` – public function that allows the user to set the shift value. This function must implement error-checking using **exceptions**: in a try/catch statement, check if (1) a value less or equal to 0 is provided; (2) a value greater than 94 is provided. In the first case, throw the error message: "Please provide a positive shift value"; In the second case, throw the error message: "Maximum shift is 94 for ASCII, please try again". The error message must be caught in a catch statement and printed to standard output, ending with a single new line (`cout << msg << endl`), another value has to be read from standard input, and the process should repeat until a valid shift is provided.

- `char encodeChar(char)` – this function implements Caesar encoding. A plaintext `char` is received as input, it is then shifted, and the shifted `char` is returned. Note that by default, the shift is set to 0. Thus, before encoding takes place, we need to make sure that the shift has been set to a valid value. Check if shift is 0, and if it is, repeat the error checking with exception handling exactly as done in `setShift(int)`. HINT: Nothing stops you from re-using `setShift(int)` code by calling it!
- `char decodeChar(char)` – this function implements Caesar decoding. A ciphertext `char` is received as input, and the decoded `char` is returned. No error-checking has to be done in this function.

The following UML illustrates the relationship between `Cipher`, `SubstitutionCipher`, and `Caesar` classes:



`Caesar` class is not an abstract class, therefore you can instantiate a `Caesar` object and test your code. Test your encoding thoroughly before going to the next step! Here are some examples for different shifts (all shifts applied to plaintext):

```
{{ Hello World! }}    – plaintext
~~#Khoor#Zruog$#!!    – ciphertext, shift = 3
++/Wt{{~/f~"{s0/--    – ciphertext, shift = 15
^^b+HOORb:RUOGcb``    – ciphertext, shift = 66
zz~Gdkkn~Vnqkc ~||    – ciphertext, shift = 94
```

Make sure you can always successfully recover the original plaintext message.

3.1.4 OneTimePad

Caesar cipher is simple, and therefore insecure: the secret message can be recovered by trying every shift [1,94] in a brute force fashion. You are going to extend the Caesar cipher and create a much stronger cipher: one-time pad. In cryptography, the one-time pad (OTP) is an encryption technique that cannot be cracked if used correctly. In this technique, a plaintext is paired with a random secret key (also referred to as a one-time pad). Then, each character of the plaintext is encrypted by combining it with the corresponding part of the secret key. If the key is truly random, is at least as long as the plaintext, is never reused in whole or in part, and is kept completely secret, then the resulting ciphertext will be impossible to decrypt or break. One-time pads were employed by Soviet espionage agencies for covert communications with agents and agent controllers.

For this assignment's approach to one-time pads, you will create a sequence of **random shifts**. Every character of plaintext will be encoded using the Caesar cipher, but the shift value for each character will be generated using a random number generator. The randomly generated shifts must obviously adhere to the range used in the `Caesar` cipher. To decipher a one-time-pad ciphertext message, the same sequence of random numbers will have to be used. Random number sequence returned by `rand()` is determined by the seed value, therefore, the seed value will have to be stored.

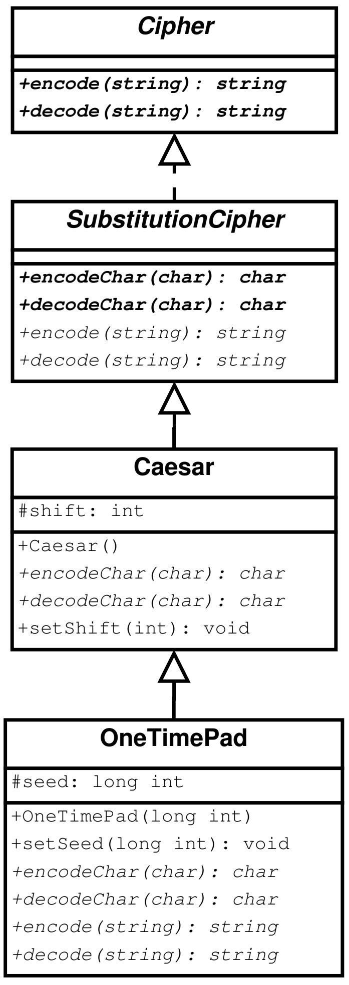
Create `OneTimePad` class in `OneTimePad.h` and `OneTimePad.cpp` files. The seed is stored in a protected member variable:

- `long int seed` – The seed value used for generating the random secret key.

`OneTimePad` adds two extra functions to the inherited interface:

- `OneTimePad(long int)` – OneTimePads' only constructor; sets the seed value to the specified seed.
- `void setSeed(long int)` – Sets the seed value to the specified seed; No error-checking is necessary.

The following UML diagram summarises the relationships between the classes.



Inherited `encodeChar()` and `decodeChar()` functions must be overidden in `OneTimePad` to implement random one-time-pad shift algorithm described above. Note that any of the inherited functions can be overidden in the derived classes as necessary, and you are free to do so.

`OneTimePad` class is not an abstract class, therefore you can instantiate a `OneTimePad` object and test your code. Test your encoding thoroughly! Here are some examples for different seeds (all one-time-pad encodings applied to plaintext):

```
{{ Hello World! }}   – plaintext
r!b`wW}ClDz%:QS55c   – ciphertext, seed = 1
18qr@Tmu\k)onOY0=0   – ciphertext, seed = 2
E,Db`WsdAhviX(mvdP   – ciphertext, seed = 9999
```

As you can see, the ciphertext produced by `OneTimePad` is much more random than `Caesar`, and there is no easy way of guessing the original text from the ciphertext. Make sure your decode function reconstructs the original text correctly!

3.1.5 Test and Submit to Fitchfork

Test your code. When you are certain it works as expected, compress all of your code (.h and .cpp) into a single archive (either .tar.gz, .tar.bz2 or .zip – make sure there are no folders or subfolders in the archive) and submit it for marking to the appropriate upload slot (Assignment 2, Task 1) before the deadline. Note that you do not need to upload a makefile for this assignment. Also note that Fitchfork is an automarking tool, not a debugger – number of uploads is limited and should be used wisely.

3.2 Task 2: Transposition cipher hierarchy (20 marks)

A transposition cipher rearranges the order of the letters in the plaintext to form the ciphertext. A predetermined method of re-ordering the letters is used, but individual letters are not changed, i.e. 'A' remains an 'A', '!' remains a '!'. One of the simplest transposition ciphers is writing the text backwards:

```
Hello World!   – plaintext
!dlroW olleH   – ciphertext
```

Obviously, such encryption is very weak, and the message is not well-hidden. A stronger type of transposition cipher arranges the letters in a matrix form:

```
H e l l
o   W o
r l d !
```

The chosen method of matrix traversal will determine the new order of letters. For example, if **columns** are concatenated to create the ciphertext, we will get the following:

```
Hello World!   – plaintext
Hore llwdlo!   – ciphertext
```

Now the meaning is obfuscated.

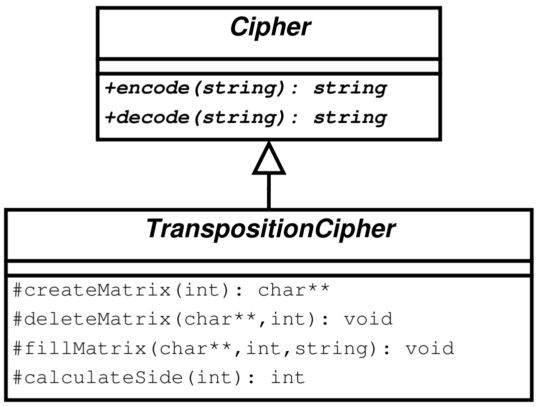
For this task, you will create a simple hierarchy of transposition ciphers, leading to two concrete ciphers: row-column and zig-zag.

3.2.1 TranspositionCipher

Create `TranspositionCipher` class in files `TranspositionCipher.h` and `TranspositionCipher.cpp`. `TranspositionCipher` inherits from `Cipher` and implements the following *protected* functions:

- `char ** createMatrix(int)` – this function creates a square (width == height) matrix of characters. The `int` parameter specifies the matrix side (width, height). Since matrix is a 2D structure, a 2D array of `char` type is dynamically allocated and returned. Characters are not initialized – this function simply creates an empty structure and returns it.
- `void deleteMatrix(char **, int)` – this function deallocates a square matrix. Assumption is made that `char**` refers to a dynamically allocated 2D array, and that the array is square. The `int` parameter specifies the side length of the array.
- `void fillMatrix(char **, int, string)` – this function receives an empty `char**` 2D array, the array side `int` (assuming the array is square), and a string. The array is filled with string characters in linear fashion, row by row, as shown in the "Hello World!" example above. **NB:** If the end of the string is reached before the end of the matrix is reached, the remainder of the matrix is filled with spaces (ASCII code 32).
- `int calculateSide(int)` – this function receives an `int` parameter corresponding to the length of a string, and derives the minimal size of a square matrix necessary to store the string. The side length of the matrix is returned. Eg: for a string of 4 characters, the minimal square matrix side is 2 (2 x 2 matrix); for a string of 5 characters, the minimal square matrix side is 3 (3 x 3 matrix).

The following UML diagram summarises the member functions of the `TranspositionCipher` class and illustrates the relationship between the `Cipher` and `TranspositionCipher` classes.

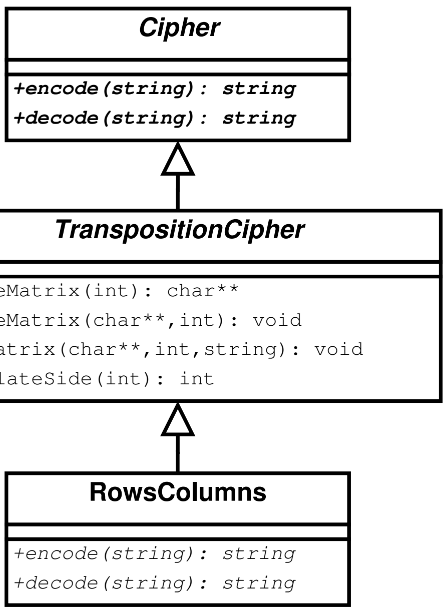


3.2.2 RowsColumns

The abstract `TranspositionCipher` class can be extended to create a concrete cipher: rows-columns. This cipher arranges the plaintext in a square matrix row-wise as previously described, and generates ciphertext by reading the matrix column-wise. To decode ciphertext, insert it into the matrix column-wise, and read it row-wise.

`RowsColumns` inherits from `TranspositionCipher`. Create `RowsColumns` class in files `RowsColumns.h` and `RowsColumns.cpp` according to the following specifications:

- `string encode(string)` – This function receives plaintext (`string`) as input, encrypts it with the help of the inherited matrix manipulation functions, and returns the ciphertext. **NB:** make sure all dynamic memory is deallocated before you exit the `encode()` function!
- `string decode(string)` – This function receives ciphertext (`string`) as input, decodes it with the help of the inherited matrix manipulation functions, and returns plaintext (`string`). If the text is not long enough to fill the corresponding minimal square matrix, this function must **throw an exception**: the error message "Incompatible text length". **NB:** make sure all dynamic memory is deallocated before you exit the `decode()` function!



Test your class thoroughly. Refer to example code for expected input/output pairs.

3.2.3 ZigZag

Zig-Zag is another variation on the same transposition cipher theme: This cipher arranges the plaintext in a square matrix row-wise as previously described. To generate ciphertext, each even (starting from 0) column is read top down, and each odd (starting from 1) column is read bottom up. If the following text is to be encrypted:

`Hello, World!..` – plaintext

The following matrix is generated:

```
H e l l
o ,   W
o r l d
! . .
```

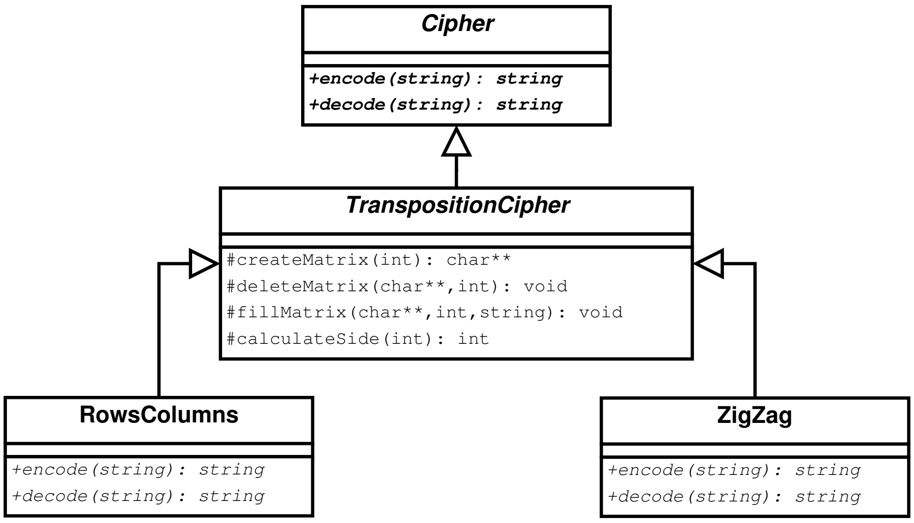
Using zig-zag encoding, the ciphertext is:

```
Hoo!.r,el l.  dWl
```

To decode the text, fill the matrix in the zig-zag fashion, and read row-wise.

`ZigZag` inherits from `TranspositionCipher`. Create `ZigZag` class in files `ZigZag.h` and `ZigZag.cpp` according to the following specifications:

- `string encode(string)` – This function receives plaintext (`string`) as input, encrypts it with the help of the inherited matrix manipulation functions, and returns the ciphertext. **NB:** make sure all dynamic memory is deallocated before you exit the `encode()` function!
- `string decode(string)` – This function receives ciphertext (`string`) as input, decodes it with the help of the inherited matrix manipulation functions, and returns plaintext (`string`). If the text is not long enough to fill the corresponding minimal square matrix, this function must **throw an exception**: the error message "Incompatible text length". **NB:** make sure all dynamic memory is deallocated before you exit the `decode()` function!



Test your class thoroughly. Refer to example code for expected input/output pairs.

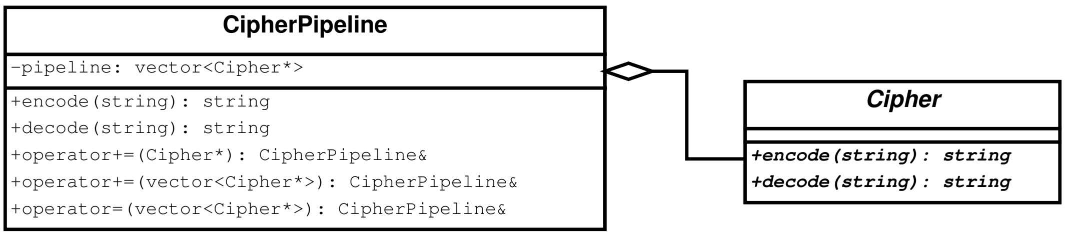
3.2.4 Test and Submit to Fitchfork

Test your code. When you are certain it works as expected, compress all of your code (.h and .cpp) into a single archive (either .tar.gz, .tar.bz2 or .zip – make sure there are no folders or subfolders in the archive) and submit it for marking to the appropriate upload slot (Assignment 2, Task 2) before the deadline. Note that you do not need to upload a makefile for this assignment. Also note that Fitchfork is an automarking tool, not a debugger – number of uploads is limited and should be used wisely.

3.3 Task 3: Cipher Pipeline (20 marks)

The ultimate cipher is a combination of many ciphers. For this task, you will create a cipher pipeline that allows to combine many different ciphers into one super-strong cipher.

Create `CipherPipeline` class in files `CipherPipeline.h` and `CipherPipeline.cpp` according to the following specifications:


CipherPipeline
-pipeline: vector<Cipher*>
+encode(string): string
+decode(string): string
+operator+=(Cipher*): CipherPipeline&
+operator+=(vector<Cipher*>): CipherPipeline&
+operator=(vector<Cipher*>): CipherPipeline&
Cipher
+encode(string): string
+decode(string): string

Class `CipherPipeline` has one private variable:

- `vector<Cipher*> pipeline` – a vector of `Cipher*` pointers.

The following functions define the public interface of `CipherPipeline`:

- `string encode(string)` – This function receives plaintext (`string`) as input and iterates through the pipeline of ciphers, applying each cipher to the text in order. The first cipher pushed onto the vector should be applied first. The resulting ciphertext is returned to the user as `string`.

- `string decode(string)` – This function receives ciphertext (`string`) as input, and decodes it using the pipeline. The last cipher in the pipeline should be applied first for successful decoding. The resulting plaintext is returned to the user as `string`.
- `CipherPipeline& operator+=(Cipher*)` – This operator adds a `Cipher*` to the end of the pipeline.
- `CipherPipeline& operator+=(vector<Cipher*>)` – This operator receives a vector of `Cipher*` pointers and adds the vector to the pipeline.
- `CipherPipeline& operator=(vector<Cipher*>)` – This operator receives a vector of `Cipher*` pointers and **replaces** the pipeline with the provided vector.

You must also implement `operator+` to allow the following syntax:

```
OneTimePad otp(12345);
ZigZag zz;

CipherPipeline pipe;
pipe = otp + zz;   // + operator is used
```

The operator should be able to take any two `Cipher` objects as inputs. It is up to you to decide where the operator is implemented. HINT: Remember that + operator can be implemented as a stand-alone function (i.e. non-member function).

Note that `CipherPipeline` has no constructor: `pipeline` will be automatically initialized using the `vector<T>` default constructor. `CipherPipeline` also **does not** provide a destructor: even though pointers are stored, the actual ciphers are never initialized within the `CipherPipeline` class, thus no assumptions can be made regarding static/dynamic allocation of the pointers.

Test and Submit to Fitchfork

Test your code. When you are certain it works as expected, compress all of your code (.h and .cpp) into a single archive (either .tar.gz, .tar.bz2 or .zip – make sure there are no folders or sub-folders in the archive) and submit it for marking to the appropriate upload slot (Assignment 2, Task 3) before the deadline. Note that you do not have to upload a makefile for this assignment. Also note that Fitchfork is an automarking tool, not a debugger – number of uploads is limited and should be used wisely.

`The End`