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

COS110 Assignment 3

Modular Cryptography

Due date: 27 September 2018, 23h30

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 such 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.

3 Your task

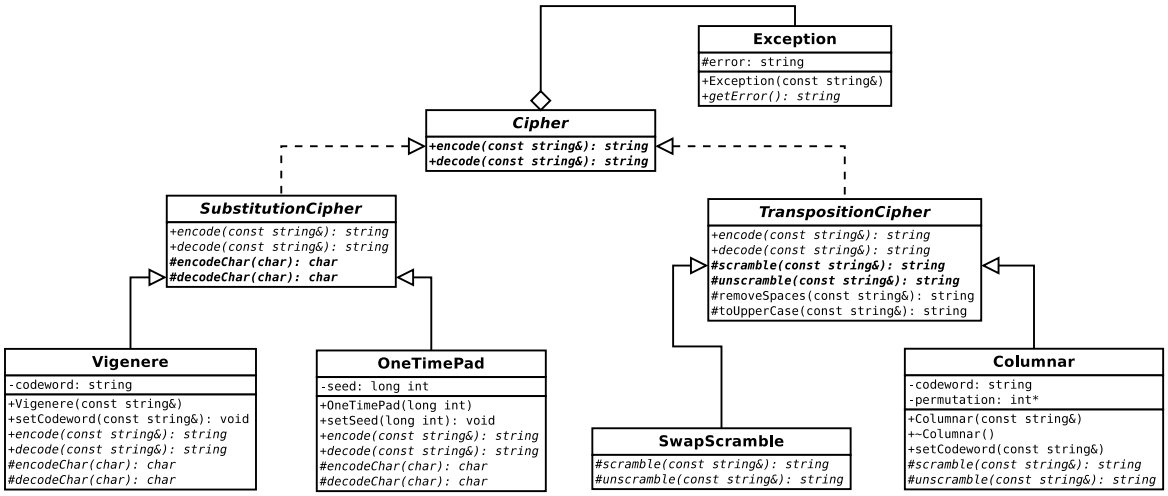
This assignment will give you an oppurtunity to work with inheritance, polymorphism, aggregation, exception handling, and cryptography. You will create multiple classes that interact

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 automarker 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, Exception, Vigenere`
2. `OneTimePad`
3. `TranspositionCipher, SwapScramble`
4. `Columnar`

The relationships between the classes are depicted in the UML below:

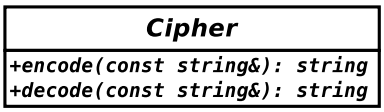


3.1 Task 1: Substitution cipher hierarchy, and the Vigenére cipher

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 hierarchy of classes representing a generic cipher interface, and implement one concrete substitution cipher: Vigenére.

3.1.1 Cipher

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

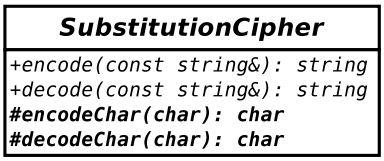
- `string encode(const string&)` – This function is abstract and must be overridden by the derived classes. The function will receive plaintext (`string`) as input, encrypt it, and return encrypted ciphertext (`string`).

- `string decode(const string&)` – This function is abstract 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 encryption that replaces individual units of plaintext with ciphertext, according to a predefined pattern. For this task, consider "units" of text to be single characters. Substitution ciphers will thus encode and decode text by encoding/decoding each character of text individually.

The `Cipher` interface can be extended to create an abstract `SubstitutionCipher` class, which will become the base class for all concrete substitution ciphers. Create the `SubstitutionCipher` class in `SubstitutionCipher.h` and `SubstitutionCipher.cpp` files, according to the UML and the specifications below.



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

- `char encodeChar(char)` – This function is abstract (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 (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 functions are protected because the user is expected to interact with the class via the inherited **public** interface of the parent class (`Cipher`). The abstract functions inherited from the parent class `Cipher` can be overidden and implemented now:

- `string encode(const 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(const string&)` – This function receives ciphertext (`string`) as input, decodes it by applying `decodeChar` to every character of ciphertext, and returns plaintext (`string`).

Note that the `SubstitutionCipher` is still an abstract class, because it has two pure virtual functions. However, it does implement some of the functionality according to the interface of the `Cipher` class. The non-abstract `encode` and `decode` functions make use of the abstract `encodeChar` and `decodeChar` functions. When a child class implements `encodeChar` and `decodeChar` as prescribed by the `SubstitutionCipher`, the `SubstitutionCipher` will already know how to apply these functions to plaintext and ciphertext. Such interaction between the classes in the hierarchy allows for modular design and distributed implementation, making it a well-known object-oriented design pattern. This design pattern is known as the *Template Method pattern*, and you are encouraged to read more about it: `https://en.wikipedia.org/wiki/Template_method_pattern`.

3.1.3 Vigenere

The Vigenére cipher is named after Blaise de Vigenére (1523-1596), and it is a simple 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. The exact "shift" for each letter is determined by a secret codeword: the distance from the first letter of the alphabet to each letter of the codeword determines the shift.

For example, suppose the secret codeword is 'BAD'. We can calculate how far each character is from the beginning of the alphabet: 'B' is the second letter in the Latin alphabet, thus it is 1 position away from the beginning ('A'). 'A' is the first letter, so the distance from the beginning is zero. Finally, 'D' is the fourth letter in the alphabet, thus the distance from 'A' is 3. Therefore, 'BAD' corresponds to the following shifts: `1, 0, 3.` Let's apply these shifts to the word 'CAT'. First, align the plaintext with the codeword and see what shifts have to be applied:

C	A	T
B	A	D
1	0	3

So 'C' has to be moved forward by 1 character. Thus, 'C' will be replaced by 'D' (next letter in the alphabet). 'A' aligns with 'A', and because 'A' is already at the beginning of the alphabet, no shift will be applied. 'T' has to be moved forward by 3 characters, and if you remember the Latin alphabet, you would know the 3rd letter after 'T' is 'W'. Thus, the encyption for 'CAT' is 'DAW':

plaintext:	C	A	T
codeword:	B	A	D
shifts:	1	0	3
ciphertext:	D	A	W

To decode, the process is reversed: ciphertext letters are aligned with the codeword, and the codeword shifts are **subtracted** from the ciphertext, i.e. the letters are shifted backwards:

ciphertext:	D	A	W
codeword:	B	A	D
shifts:	1	0	3
plaintext:	C	A	T

What if your text is longer than the codeword? Well, you can always just apply the codeword repetitively:

ciphertext:	H	E	L	L	O
codeword (repeated):	B	A	D	B	A
shifts:	1	0	3	1	0
plaintext:	I	E	O	M	O

'IEOMO' sounds nothing like 'HELLO', thus you can consider the text to be encrypted.

In C++, all characters correspond to integer ASCII values, which makes `char` type very convenient to work with: to shift a character upwards or backwards, you can simply add the necessary integer shift value to the character, or subtract a shift value from the character. You are also allowed to subtract the characters from each other, or add them to one another. The following code is valid in C++:

```
char plainChar = 'H';
int shift = 2;
char cipherChar = plainChar + shift;  // encrypt a character
char decodeChar = cipherChar - shift; // decode a character
```

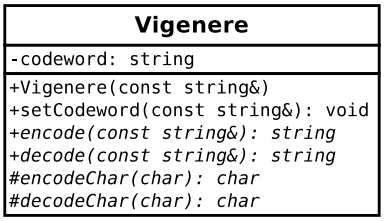
The table below lists the ASCII codes of all printable characters:

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. Space character (ASCII value 32) will be treated as the first letter of the alphabet.

What will happen if you try to shift '}' (ASCII code: 125) by 3 positions forward? 125 + 3 = 128, which is out of the printable 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 '}'.

Implement the `Vigenere` class according to the UML and the specifications below:

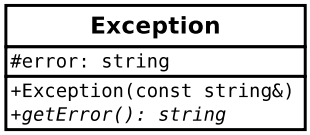


The `Vigenere` class has a private member `codeword`:

- `string codeword` – The codeword used for encoding/decoding, as described earlier.

The `Vigenere` class has the following member functions:

- `Vigenere(const string&)` – Parameterised constructor that sets the codeword to the provided string.
- `void setCodeword(const string&)` – public function that allows the user to set the codeword to another string. This function must handle invalid input using **exceptions**. For this purpose, create the `Exception` class in a file called `Exception.h` according to the following UML:



The `Exception` object is initialised with an error message, and provides the `getError()` function to return the error message. You will be using various error messages for various exceptions throughout this assignment. Thus, you may consider extending the `Exception` class for various types of exceptions. However, please note that **all** `Exception`-related code should be placed in `Exception.h`. **DO NOT** create a separate .cpp file.

The codeword should only be accepted if it is more than one character long, and if it is not

made entirely of ' ' (space) characters. The space character (ASCII code 32) is the first printable character in the given range, thus a codeword made entirely of spaces would result in shifts of zero, and the text will not be encrypted. If the codeword is either too short, or made out of spaces, throw an `Exception` object initialised with the following message: "The codeword provided is not going to generate a safe encryption". The message should contain **no punctuation** and **no new line characters**.

- `char encodeChar(char)` – this function implements Vigenére encoding. A plaintext `char` is received as input and shifted according to the current place in the codeword. The shifted `char` is returned. No error-checking has to be done in this function.
- `char decodeChar(char)` – this function implements Vigenére decoding. A ciphertext `char` is received as input, and the decoded `char` is returned. No error-checking has to be done in this function.
- Note that `encode(string)` and `decode(string)` are also listed in the UML. You *may* override these functions if you need to add extra functionality to them. Remember that child classes have access to the base class implementations for the purpose of code re-use.

`Vigenere` class is not an abstract class, therefore you can instantiate a `Vigenere` object and test your code. Test your encoding thoroughly before submitting for marking! Here are some examples for different codewords:

```
{{ Hello World! }}    – plaintext
>=Ni4./1Nx>4/&OAL?    – ciphertext, codeword = "BANANA"
oac;Ka‘UcJUg‘Jdrcr    – ciphertext, codeword = "secret"
I]s=K_:Yn<=T‘YfrKg    – ciphertext, codeword = "MasterMind"
```

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

3.1.4 Test and submit for marking

Test your code. When you are certain the code 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 3, Task 1) before the deadline. Note that you do not need to upload a makefile for this assignment. Also note that the number of uploads is limited and should be used wisely.

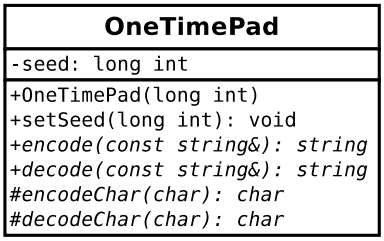
3.2 Task 2: One-Time Pad cipher

Vigenére cipher is simple, and therefore not very secure, especially if a short and simple codeword is used. You are going to create a much stronger substitution cipher: the 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 random sequence of shifts is used in place of a codeword. If the sequence 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 decode 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 similarly to the Vigenére cipher, but the shift value for each character will be generated using a random number generator. The randomly generated shifts must adhere to the `[1,94]` range (inclusive), and should wrap around the ASCII table in the same fashion as used in the `Vigenere` 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()` (`http://www.cplusplus.com/reference/cstdlib/rand/`) is determined by the seed value, therefore, the seed value will have to be stored.

Create the `OneTimePad` class in `OneTimePad.h` and `OneTimePad.cpp` files. Implement the `OneTimePad` class according to the specifications and the UML below:



The seed is stored in a private member variable:

- `long int seed` – The seed value used to generate the random shifts.

`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. Only non-negative values can be accepted. If a negative value is given as input, throw an `Exception` object initialised with the following message: "Negative number provided".

Inherited `encodeChar()` and `decodeChar()` functions must be overidden in the `OneTimePad` to implement random one-time-pad shift algorithm as 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 (random number generators are platform-dependent, and encryptions may differ on your machine):

```
{{ 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 from the examples, `OneTimePad` is a lot more random than `Vigenere`, which makes it a stronger cipher.

3.2.1 Test and submit for marking

Test your code. When you are certain the code 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 3, Task 2) before the deadline. Note that you do not need to upload a makefile for this assignment. Also note that the number of uploads is limited and should be used wisely.

3.3 Task 3: Transposition cipher hierarchy, and the Scramble cipher

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:

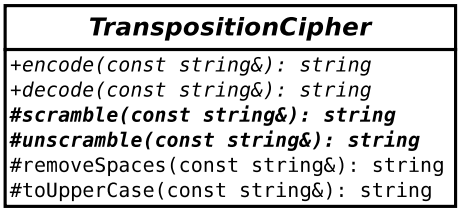
`Hello World!` – plaintext
`!dlroW olleH` – ciphertext

Obviously, such encryption is very weak, and the message is not well-hidden. Stronger transposition ciphers create permutations of characters that make it a lot harder to guess the original text. For example, it is common practice to remove all spaces from the plaintext, as well as to make the text upper-case. This way, it becomes harder to guess the letters associated with the beginning of words and sentences. Once the spaces are removed, and the text is capitalised, you will not be able to recover the original spacing and/or capitalisation. However, that is a small price to pay for the safety of secret information!

For this task, you will create a simple hierarchy of transposition ciphers, and implement a concrete transposition cipher: swap-scramble.

3.3.1 TranspositionCipher

The `Cipher` interface can be extended to create the abstract `TranspositionCipher` class. Create the `TranspositionCipher` class in files `TranspositionCipher.h` and `TranspositionCipher.cpp`. Implement the `TranspositionCipher` class according to the following specifications and UML:



`TranspositionCipher` declares two **protected** abstract functions:

- `string scramble(const string&)` – This function is abstract (pure virtual) and must be overridden by the derived classes. The function will receive a plaintext string as input, encrypt it using a transposition algorithm, and return the ciphertext.
- `string unscramble(const string&)` – This function is abstract (pure virtual) and must be overridden by the derived classes. The function will receive a ciphertext string as input, decode it according to a transposition algorithm, and return the plaintext.

As mentioned earlier, transposition ciphers can be strengthened by removing all spaces from plaintext, as well as converting the plaintext to uppercase. Two **protected** functions can be added to the `TranspositionCipher` class to perform these string manipulations:

- `string removeSpaces(const string&)` – This function receives string input, and returns a copy of the input with all spaces removed.
- `string toUpperCase(const string&)` – This function receives string input, and returns a copy of the input with all characters converted to uppercase.

The functions are protected because the user is expected to interact with the class via the inherited public interface of the parent class (`Cipher`). The abstract functions inherited from the parent class `Cipher` can be overidden and implemented now:

- `string encode(const string&)` – This function receives plaintext (`string`) as input, removes the spaces from plaintext, converts plaintext to uppercase, and finally applies the `scramble` function to the plaintext, returning the ciphertext (`string`). This function must handle invalid input using **exceptions**. No amount of transposition will be able to encrypt a message which consists of only one character. Therefore, if the given plaintext is shorter

than 2 characters, throw an `Exception` object initialised with the following message: "The provided text is too short to be safely encrypted".

- `string decode(const string&)` – This function receives ciphertext (`string`) as input, decodes it by applying `unscramble`, and returns the resulting plaintext (`string`).

Similarly to the `SubstitutionCipher` class, the `TranspositionCipher` class makes use of the *Template Method* design pattern.

3.3.2 SwapScramble

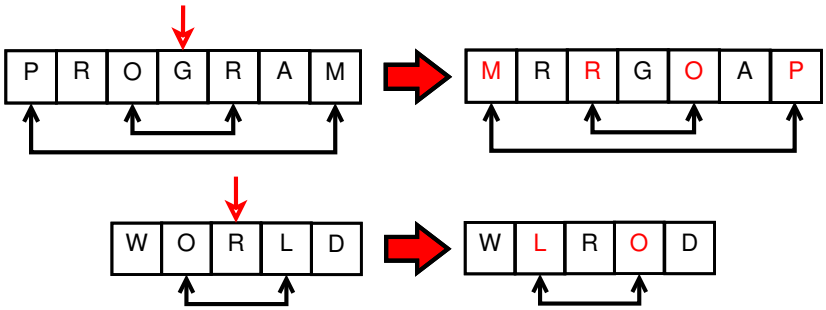
The abstract `TranspositionCipher` class can be extended to create a concrete cipher: swap-scramble. This cipher re-arranges the plaintext by locating the midpoint of the plaintext, and swapping every second pair of letters on either side of the midpoint. The swap-scramble cipher is similar to reversing the text, but because only every second letter is reversed, the original text is not as easy to recover.

The following swapping rules apply:

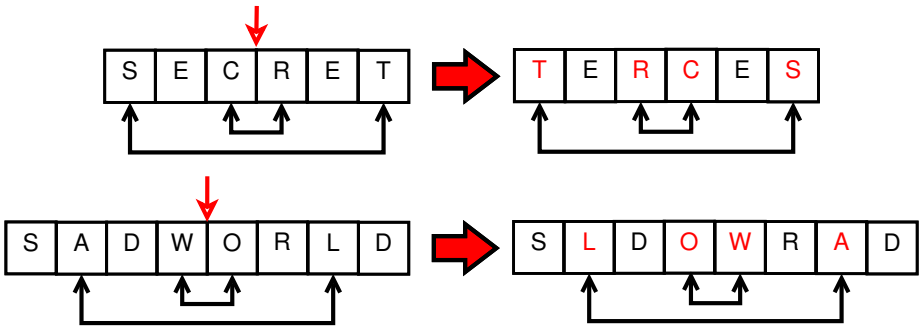
1. If the number of characters in plaintext is **odd**, the middle letter at index `i = size/2` remains stationary, and characters at locations `i+1` and `i-1` are swapped, followed by the swap of characters at locations `i+3` and `i-3`, etc., until the end of the string is reached.
2. If the number of characters in plaintext is **even**, the letter at index `i = size/2` is swapped with `i-1`, followed by the swap of characters at locations `i+2` and `i-3`, etc., until the end of the string is reached.

The following examples illustrate the swapping rules:

1. Odd text length:



2. Even text length:



The same algorithm can be used for both encoding and decoding.

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

SwapScramble
#scramble(const string&): string *#unscramble(const string&): string*

- `string scramble(const string&)` – This function receives plaintext (`string`) as input, encrypts it using the swap-scramble algorithm described above, and returns the ciphertext.
- `string unscramble(const string&)` – This function receives ciphertext (`string`) as input, decodes it using the swap-scramble algorithm described above, and returns the plaintext.

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

3.3.3 Test and submit for marking

Test your code. When you are certain the code 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 3, Task 3) before the deadline. Note that you do not need to upload a makefile for this assignment. Also note that the number of uploads is limited and should be used wisely.

3.4 Task 4: Columnar cipher

Another way to scramble the plaintext is to arrange the letters in a table, and to construct the ciphertext by concatenating table **columns**. For example, if the plaintext is "HELLO,WORLD!", you can fill a table row by row:

```
H E L L
O , W O
R L D !
```

To generate the ciphertext, read the table column by column:

```
HORE,LLWDLO!
```

This algorithm is simple, but it does not hide the original text very well. We can make the cipher stronger by adding a secret codeword that will determine the **order** in which the columns are read. For example, suppose the codeword is "MOLO". There are 4 letters in the codeword, so we will create 4 columns and add as many rows as necessary to fit in the plaintext:

M	**O**	**L**	**O**
H	E	L	L
O	,	W	O
R	L	D	!

We will now assign numbers to the codeword letters according to their alphabetical order. 'L' is the first one alphabetically, so it will get the smallest number: 1. 'M' follows 'L' alphabetically, thus we can make it number 2. 'O' is the last one alphabetically, therefore it is assigned number 3. Add the numbers to the columns:

M	**O**	**L**	**O**
2	**3**	**1**	**3**
H	E	L	L
O	,	W	O
R	L	D	!

Now that the columns have been numbered, we can read the text columns-wise according to the assigned numbers: first the 'L' column, then the 'M' column, and finally both 'O' columns. To make the cipher stronger still, a special rule will be imposed on the columns that have the **same number**: the text will be read **row-wise** from the set of columns that share the same index. The resulting ciphertext is then:

`LWDHOREL,OL!`

(1st column: LWD, 2nd column: HOR, 3rd and 4th columns: EL,OL!, where the blue characters correspond to the second 'O' column.)

Now the plaintext has been safely scrambled. How do you recover the plaintext from the ciphertext? Well, if you know the codeword, the task is easy: create a table of the correct size, use the codeword indices to fill the table in the correct order. Once the table has been filled, read the text row-wise. Let's decode the ciphertext generated earlier:

`LWDHOREL,OL!`

We know that the codeword is 'MOLO'. If you divide the length of the ciphertext by the length of the codeword, you will get the number of rows necessary to store the text: `12/4 = 3`. Create a `3x4` table:

M	**O**	**L**	**O**
2	**3**	**1**	**3**

We know that the 'L' column was read first, thus we will put the first three characters in that column:

M	**O**	**L**	**O**
2	**3**	**1**	**3**
		L	
		W	
		D	

Fill the second column with the next 3 ciphertext characters:

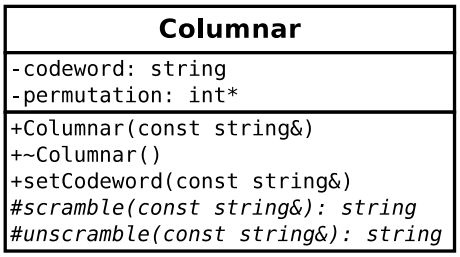
M	**O**	**L**	**O**
2	**3**	**1**	**3**
H		L	
O		W	
R		D	

The remaining two columns have the same index (3), thus they will be filled in a row-wise fashion (blue, followed by red, followed by green):

M	**O**	**L**	**O**
2	**3**	**1**	**3**
H	E	L	L
O	,	W	O
R	L	D	!

If we read this table row-wise, we will recover the plaintext in its original form.

The `Columnar` class inherits from the `TranspositionCipher`. Create the `Columnar` class in files `Columnar.h` and `Columnar.cpp` according to the UML and the specifications below.



The `Columnar` class has two private members: a `string` codeword, and a dynamic 1D array of integers (`permutation`). Use the integer array to store indices associated with the given keyword. For the 'MOLO' keyword, the array of indices would be `[2,3,1,3]`. You may also decide to start the index count at 0, in which case the values would be `[1,2,0,2]`. Note that storing the indices in a separate array is recommended, but not required.

- `Columnar(const string&)` – The only constructor of the `Columnar` class. The provided string input is used to set the keyword and the corresponding index permutaion.
- `~Columnar()` – Destructor of the `Columnar` class. Deallocated the dynamic integer array.
- `void setCodeword(const string&)` – This function sets the keyword to the provided string input. Invalid input should be handled using **exceptions**: if the provided codeword is shorter than 2 characters long, throw an `Exception` object initialised with the following message: "The codeword provided is not going to generate a safe encryption". Imagine that the provided codeword is 'AAA': since all characters are the same, the ciphertext would be read row-wise, and no encryption will happen. To prevent this, check if any two adjacent characters in the codeword are the same. If they are, throw an `Exception` object initialised with the following message: "The codeword provided is not going to generate a safe encryption". The following codewords should all generate an exception: 'A', 'AAA', 'DOOM'.
- `string scramble(const string&)` – This function receives plaintext (`string`) as input, encrypts it using the columnar cipher described above, and returns the ciphertext. **NB:** If the end of the plaintext is reached before the end of the table is reached, the remainder of the table should be filled with character 'A' (ASCII code 65).
- `string unscramble(const string&)` – This function receives ciphertext (`string`) as input, decodes it using the columnar cipher described above, and returns the plaintext. Errors should be handled using **exceptions**: if the provided ciphertext is not long enough to fill the corresponding table without any gaps, the function must throw an `Exception` object initialised with the following message: "The ciphertext is of incorrect length"

HINT: The columnar cipher relies on putting the text into a table/matrix, and reading from that table/matrix in a particular way. You can make the implementation a lot simpler by adding extra helper functions to the `Columnar` class to assist you with creating and deleting 2D arrays. For example, you may consider writing a private `createMatrix(int, int)` function to allocate a 2D array of the given dimensionality; it will also be convenient to have a `deleteMatrix(char **, int)` function to deallocate the 2D array, and a `fillMatrix(char **, int, int, string)` function to fill the empty matrix with the characters of the given string. None of these functions are required, they will not be tested by the automarker. However, you will find that working with modular code is a lot easier!

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

3.4.1 Test and submit for marking

Test your code. When you are certain the code 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 3, Task 4) before the deadline. Note that you do not need to upload a makefile for this assignment. Also note that the number of uploads is limited and should be used wisely.

The End