Assignment 3

COS 110

Babushka Encryption

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

Department of Computer Science
Deadline: 29 October 2021 at 23:00

Objectives:

- Become familiar with C++ Polymorphism and Exception handling.

General instructions:

- This assignment should be completed individually, **no group effort** is allowed.
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not include any C++ libraries that are not stated in this specification. Doing so will result in a mark of zero.
- If your code does not compile you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- **All submissions will be checked for plagiarism.**
- Read the entire assignment before you start coding.
- You will be afforded three upload opportunities.

Plagiarism:

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to `http://www.library.up.ac.za/plagiarism/index.htm` (from the main page of the University of Pretoria site, follow the *Library* quick link, and then choose the *Plagiarism* option under the *Services* menu). If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding. Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

Overview

Babushka dolls (otherwise known as Matryoshka dolls or Russian nesting dolls) are dolls which get progressively larger in size. Each larger doll encapsulates a doll of a smaller size. In this assignment, the Babushka analogy is used to create an encryption and decryption scheme.

Figure 1: Babushka dolls.

What is encryption and decryption?

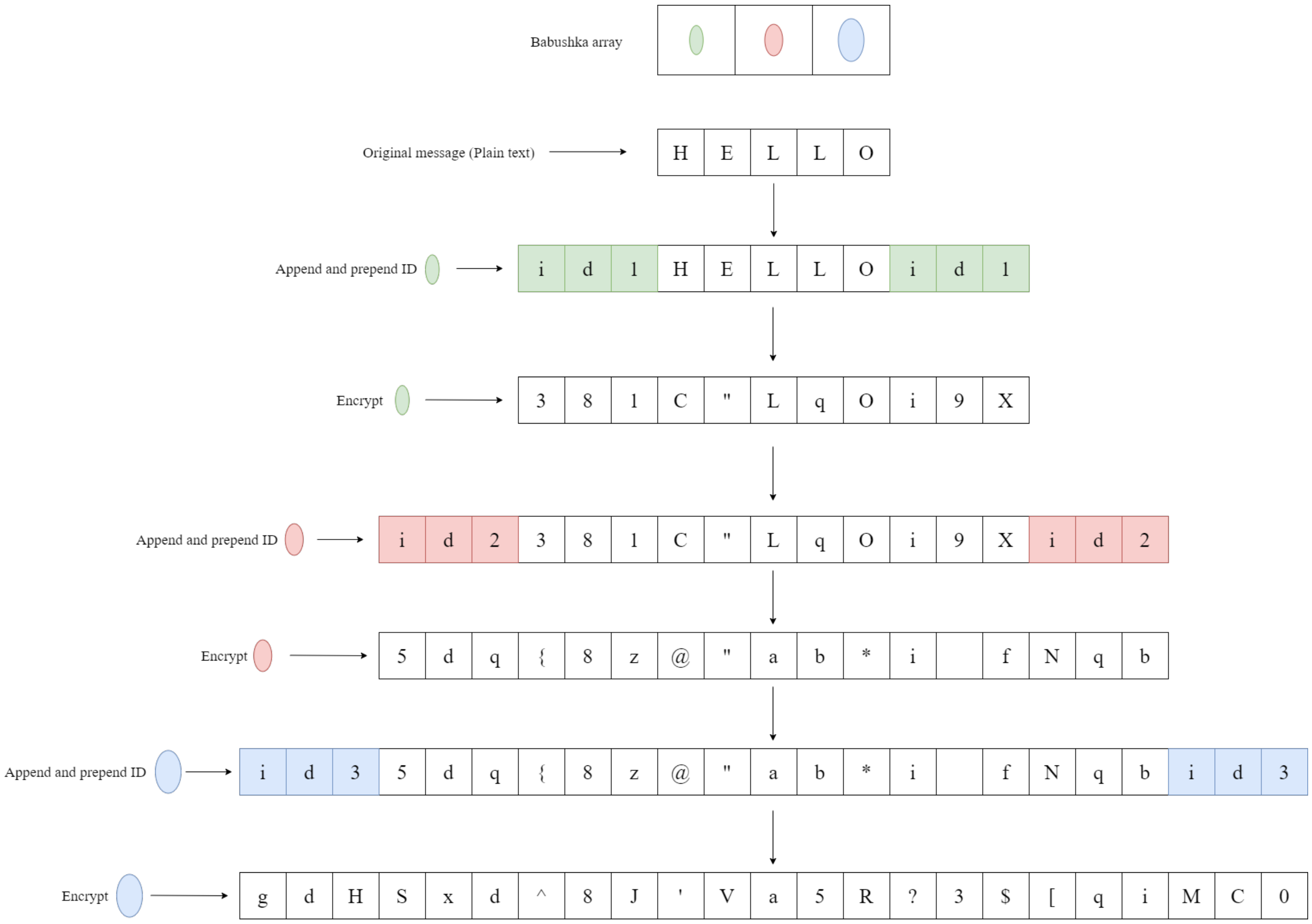
Encryption, for the purposes of this assignment, involves transforming plain text into cipher text. Plain text is simply any message that a human can read and understand in its original form. Cipher text is the result of encryption, which makes it unreadable for humans. Given a message X, encryption transforms X into cipher text C. The difference between X and C is that X is readable by humans whereas C is not. Decryption is the process of going backwards from cipher text C back to plain text X. Encryption usually makes use of a secret key. The key is used to manipulate the plain text and transform it into cipher text in a **reversable** way dependant on the key. Therefore, only if you have a key that was used for encryption, can you successfully decrypt a message (similar to how keys work in real life). This assignment is highly oversimplified so please don't consider your code as actual cryptography.

In this assignment, arrays of type `unsigned char` contain data which is to be encrypted or decrypted based on different strategies. The arrays go through multiple stages of encryption similar to how a Babushka doll contains all smaller dolls than itself. It is important to note that `unsigned char`s hold integer values in the range 0 to 255. These values correspond to certain readable characters defined by the ASCII representation. Please see how decimal numbers correspond to each character here: `https://www.asciitable.com/`. In this assignment we will be working only between the decimal range 32 to 126 (inclusive) which corresponds to all characters between the *space* character and the tilde ˜ character (inclusive).

Remember the following: `unsigned char`s store numbers, however you can print their character representation by simply using `cout << x` where `x` is the `unsigned char`. By manipulating the number an `unsigned char` stores, the character it represents will also change.

Encryption visualisation

A Babushka in this assignment has the ability to encrypt and decrypt. Each Babushka has its own colour, and each colour has its own method of encrypting and decrypting. Furthermore, each Babushka has its own id which is essentially its *key* for encryption and decryption. A Controller class contains an array of Babushka objects, and in order for encryption to occur, each Babushka object in the array is used one after the other to progressively encrypt a particular message. Decryption is the reverse of this process (i.e. given a cipher text as input, use the Babushkas to decrypt and obtain the original plain text). In Figure 2, an array with a green, red and blue Babushka is defined. Each Babushka has its own id which is set to id1, id2 and id3 for each respective Babushka (in the actual assignment you will be given these ids). Given the Babushka array and some plain text, each Babushka gets a chance to append and prepend its id to the input and then proceeds to use its particular type of encryption. Each Babushka is used from left to right in the array for encryption. The output of all previous Babushkas is used as input to the current Babushka (this is why they are called nesting dolls). The output after the final Babushka has encrypted is the final cipher text.



H E L L O

i d 1 H E L L O i d 1

3 8 1 C " L q O i 9 X

i d 2 3 8 1 C " L q O i 9 X i d 2

5 d q { 8 z @ " a b * i f N q b

i d 3 5 d q { 8 z @ " a b * i f N q b i d 3

g d H S x d ^ 8 J ' V a 5 R ? 3 $ [q i M C 0

Figure 2: Babushka encryption example. Note that the results of the encryption here are made up and are just an example.

Decryption visualisation

You need to figure this out yourself. Decryption simply needs to undo the encryption result of encryption. Each Babushka also has a decrypt function.

Knowledge requirements and Important notes

In order to accomplish this assignment, you must understand inheritance, polymorphism and exceptions. Furthermore, you need to be able to interpret UML class diagrams and translate the relevant information into your code. Note that functions that are in italics in this specification represent **pure virtual functions**, these functions are abstract and must be implemented by child classes. Note that some of the diagrams are also given as extra image files because they may be hard to view in the PDF. **All inheritance in this assignment is public inheritance**. All custom exceptions thrown are not dynamic memory (they are objects on the stack), and they should be caught by reference. The arrays of `unsigned char` do not contain null termination, they are simply arrays of specific characters (not cstrings).

Babushka Classes

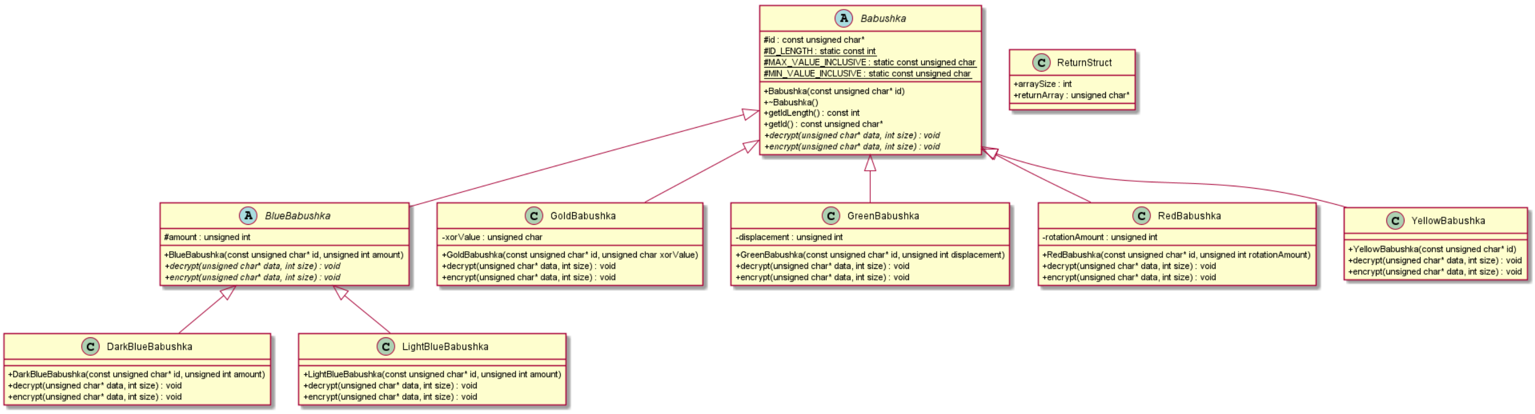

Babushka
#id : const unsigned char*
#ID_LENGTH : static const int
#MAX_VALUE_INCLUSIVE : static const unsigned char
#MIN_VALUE_INCLUSIVE : static const unsigned char
+Babushka(const unsigned char* id)
+~Babushka()
+getIdLength() : const int
+getId() : const unsigned char*
+decrypt(unsigned char* data, int size) : void
+encrypt(unsigned char* data, int size) : void
ReturnStruct
+arraySize : int
+returnArray : unsigned char*
BlueBabushka
#amount : unsigned int
+BlueBabushka(const unsigned char* id, unsigned int amount)
+decrypt(unsigned char* data, int size) : void
+encrypt(unsigned char* data, int size) : void
GoldBabushka
-xorValue : unsigned char
+GoldBabushka(const unsigned char* id, unsigned char xorValue)
+decrypt(unsigned char* data, int size) : void
+encrypt(unsigned char* data, int size) : void
GreenBabushka
-displacement : unsigned int
+GreenBabushka(const unsigned char* id, unsigned int displacement)
+decrypt(unsigned char* data, int size) : void
+encrypt(unsigned char* data, int size) : void
RedBabushka
-rotationAmount : unsigned int
+RedBabushka(const unsigned char* id, unsigned int rotationAmount)
+decrypt(unsigned char* data, int size) : void
+encrypt(unsigned char* data, int size) : void
YellowBabushka
+YellowBabushka(const unsigned char* id)
+decrypt(unsigned char* data, int size) : void
+encrypt(unsigned char* data, int size) : void
DarkBlueBabushka
+DarkBlueBabushka(const unsigned char* id, unsigned int amount)
+decrypt(unsigned char* data, int size) : void
+encrypt(unsigned char* data, int size) : void
LightBlueBabushka
+LightBlueBabushka(const unsigned char* id, unsigned int amount)
+decrypt(unsigned char* data, int size) : void
+encrypt(unsigned char* data, int size) : void

Figure 3: Class hierarchy for Babushkas.

Each `Babushka` subclass performs a certain encryption/decryption operation. Encryption and decryption in `Babushka` classes operate on arrays of type `unsigned char` and modify the values stored in these arrays (i.e. they do not copy these arrays, they simply operate on them and transform them).

Babushka

An abstract base class which is the base class of all Babushkas. Here are some notes:

- The `id` member variable is an array of type `unsigned char` and contains alphanumeric characters. The constructor of this class sets this member variable without creating any new memory. The `ID_LENGTH` member variable is the size of this array.
- The `ID_LENGTH`, `MAX_VALUE_INCLUSIVE` and `MIN_VALUE_INCLUSIVE` member variables are static constants. They are initialised in the `Babushka.h` header file to the constants 10, 126 and 32 respectively. This implies that all `id`s will have length 10 in this assignment.
- The `MAX_VALUE_INCLUSIVE` and `MIN_VALUE_INCLUSIVE` are bounds which will are relevant for some of the subclasses of this class.
- The ~`Babushka()` destructor must free the `id` array memory if it exists. Note this is a **virtual destructor**!
- The `getIdLength` function returns the `ID_LENGTH`.
- The `getId` function returns the `id` member variable.
- `encrypt` and `decrypt` are pure virtual functions and do not have an implementation in this class. Each `encrypt` and `decrypt` accepts a `data` array and the `size` of the array as a parameter.

BlueBabushka

This is a subclass of `Babushka`, however this class still does not implement `encrypt` or `decrypt` because it has two child classes of its own. `BlueBabushka` encryption involves performing an addition or subtraction of a fixed amount for each element in a provided array. More details are explained in the `LightBlueBabushka` and `DarkBlueBabushka`. Here are some notes:

- The constructor accepts the `Babushka id` which must be forwarded to the base class `Babushka` constructor. This class receives an additional `amount` parameter which is set to the member variable `amount`.
- `encrypt` and `decrypt` are pure virtual functions and do not have an implementation in this class.

DarkBlueBabushka

This is a subclass of `BlueBabushka` and implements the `encrypt` and `decrypt` functions. Here are some notes:

- The constructor accepts the `Babushka id` and an `amount` parameter which are both forwarded to the base class `BlueBabushka` constructor.
- The `encrypt` function is implemented in this class and modifies the elements of the `data` array. The function works as follows: **subtract** the `amount` member variable from each element in the provided `data` array. If any value in the array goes above `MAX_VALUE_INCLUSIVE` then an `OverflowException` object must be thrown. If any value in the array goes below `MIN_VALUE_INCLUSIVE` then an `UnderflowException` object must be thrown. These exception classes are explained later in this specification. The exceptions are not dynamic memory (they are objects on the stack).
- The `decrypt` function is should do the inverse of the `encrypt` function of this class. This function should also throw and `OverflowException` or `UnderflowException` if the same situation occurs as described in the `encrypt` function of this class.

LightBlueBabushka

This is a subclass of `BlueBabushka` and implements the `encrypt` and `decrypt` functions. Here are some notes:

- This class operates almost exactly the same as `DarkBlueBabushka`.
- The `encrypt` function has a slight change from `DarkBlueBabushka`: instead of **subtracting** the `amount` member variable from each element in the `data` array, you must **add** it to each element in the `data` array. Throw an `OverflowException` or `UnderflowException` for the same reasons as explained previously.
- The `decrypt` function is should do the inverse of the `encrypt` function of this class. This function should also throw and `OverflowException` or `UnderflowException` when appropriate.

GoldBabushka

This is a subclass of `Babushka` and implements the `encrypt` and `decrypt` functions. Here are some notes:

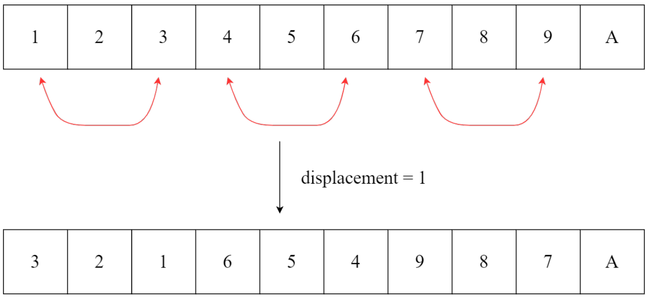
- The constructor accepts the `Babushka id` and a `xorValue` parameter. The `Babushka id` is forwarded to the `Babushka` constructor. The `xorValue` is used to set the member variable of this class.

- The `encrypt` function is implemented in this class and modifies the elements of the `data` array. The function works as follows: perform a bitwise *Exclusive Or (XOR)* operation between each element in the provided `data` array and the `xorValue` member variable. If any value in the array goes above `MAX_VALUE_INCLUSIVE` then an `OverflowException` object must be thrown. If any value in the array goes below `MIN_VALUE_INCLUSIVE` then an `UnderflowException` object must be thrown. These exception classes are explained later in this specification.
- Hint: lookup the bitwise XOR operator in C++.
- The `decrypt` function is should do the inverse of the `encrypt` function of this class. This function should also throw and `OverflowException` or `UnderflowException` if the same situation occurs as described in the `encrypt` function of this class. Hint: the inverse of the XOR is simply XOR!

GreenBabushka

This is a subclass of `Babushka` and implements the `encrypt` and `decrypt` functions. Here are some notes:

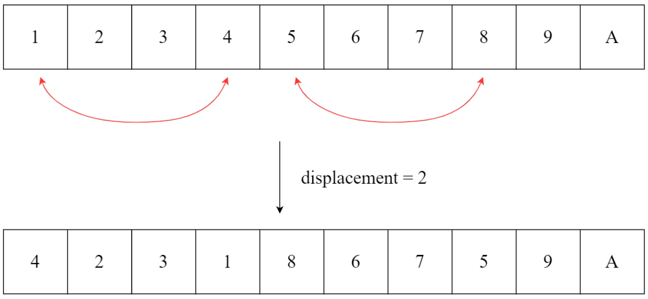
- The constructor accepts the `Babushka id` and a `displacement` parameter. The `Babushka id` is forwarded to the `Babushka` constructor. The `displacement` is used to set the member variable of this class.
- The `encrypt` function is implemented in this class and modifies the elements of the `data` array. A displacement operation needs to occur which makes use of the `displacement` member variable. A displacement operation iterates over the array and swaps elements which are a certain `displacement` from one another. You may assume that `displacement` will always be greater than zero. At least one `displacement` operation should occur. If the `displacement` member variable is too large such that not even a single displacement operation can occur, then throw a `DisplacementException`. Please make use of the following example:



1	2	3	4	5	6	7	8	9	A

3	2	1	6	5	4	9	8	7	A

Figure 4: Example transformation for `displacement` equal to one.



1	2	3	4	5	6	7	8	9	A

4	2	3	1	8	6	7	5	9	A

Figure 5: Example transformation for `displacement` equal to two.

- Exceptions are explained later in this specification.
- The `decrypt` function does the inverse of the `encrypt` function of this class. It should also throw a `DisplacementException` if it at least one displacement operation can not occur.

RedBabushka

This is a subclass of `Babushka` and implements the `encrypt` and `decrypt` functions. Here are some notes:

- The constructor accepts the `Babushka id` and a `rotationAmount` parameter. The `Babushka id` is forwarded to the `Babushka` constructor. The `rotationAmount` is used to set the member variable of this class.
- The `encrypt` function is implemented in this class and modifies the elements of the `data` array. A right rotation operation needs to occur which makes use of the `rotationAmount` member variable. A right rotation operation rotates the last `rotationAmount` number of elements from the end of the `data` array to the beginning of the `data` array. You may assume that `rotationAmount` will always be greater than zero. If the `rotationAmount` is greater than or equal to the size of the `data` array, then throw a `RotateException`.

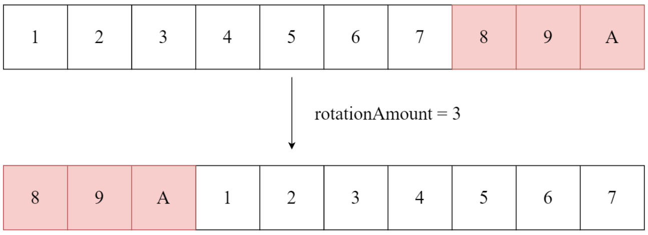


Figure 6: Example transformation for `rotationAmount` equal to three.

- Exceptions are explained later in this specification.
- The `decrypt` function does the inverse of the `encrypt` function of this class. It should also throw a `RotateException` for the same reason as the `encrypt` function.

YellowBabushka

This is a subclass of `Babushka` and implements the `encrypt` and `decrypt` functions. Here are some notes:

- The constructor accepts the `Babushka id`. The `Babushka id` is forwarded to the `Babushka` constructor.
- The `encrypt` function is implemented in this class and modifies the elements of the `data` array. This class simply reverses the elements in the `data` array.
- The `decrypt` does the inverse of the `encrypt` function.

ReturnStruct struct

The `ReturnStruct` struct is used to wrap an array and its size together. This allows a function to return two values inside a struct. The `.h` file for this struct is provided (and there is no need for a `.cpp` file). Consult Figure 3 for how this struct is defined. `ReturnStruct` is used later on in this specification.

BabushkaException classes

All `BabushkaExceptions` inherit from the `BabushkaException` class. Here are some notes:

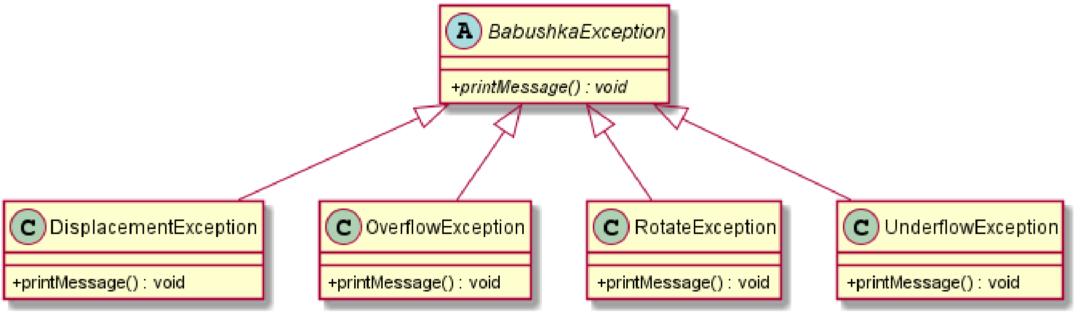


Figure 7: Class hierarchy for BabushkaExceptions.

- `BabushkaException` has a pure virtual function called `printMessage` which must be implemented by its child classes. There should be no `.cpp` file for the `BabushkaException` class because it does not have any specific implementation. Note these are normal classes which **do not** inherit from the C++ `<exception>` library.
- The subclasses of `BabushkaException` implement the `printMessage` function. The `printMessage` function simply prints a string followed by an `endl`. These are the strings to be printed for each class `DisplacementException`, `RotateException`, `OverflowException`, `UnderflowException` respectively:
  - `displacement exception occured`
  - `rotation exception occured`
  - `overflow exception occured`
  - `underflow exception occured`

ControllerException Class

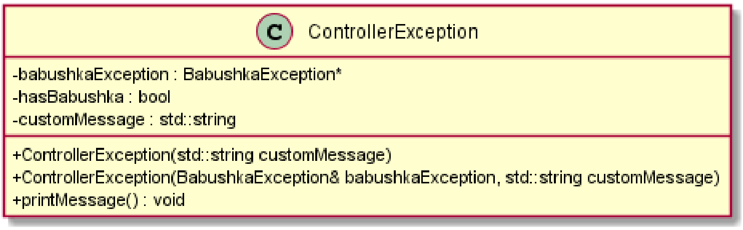
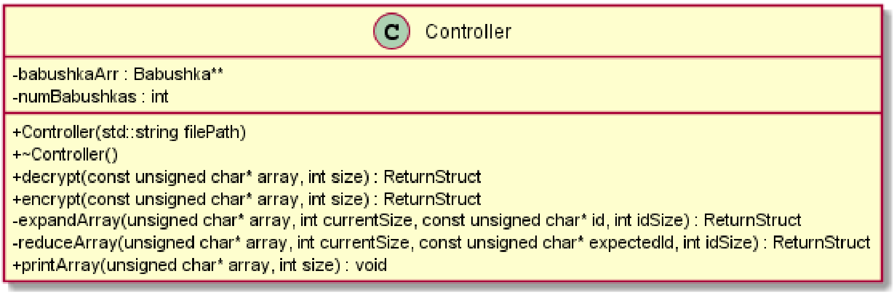

C ControllerException
-babushkaException : BabushkaException*
-hasBabushka : bool
-customMessage : std::string
+ControllerException(std::string customMessage)
+ControllerException(BabushkaException& babushkaException, std::string customMessage)
+printMessage() : void

Figure 8: Controller class UML diagram.

A `ControllerException` is what is thrown from the `Controller` class. This contains a `customMessage` to be printed as well as an optional `BabushkaException` which is wrapped inside this object as a member variable. This class **does not** inherit from C++ `<exception>`. Here are some notes:

- This class has two constructors for different use cases. The constructor that accepts just a string simply sets the `customMessage` member variable and sets `hasBabushka` to `false`.
- The second constructor accepts both a `BabushkaException` and a `customMessage`. The corresponding member variables should be set, and the `hasBabushka` member variable should be set to `true`.
- The `printMessage` function should do the following:
  - Print the `customMessage` member variable followed by an `endl`.
  - If a `BabushkaException` exists in this class, then its message should be printed by simply calling the `printMessage` function on the object.

Controller class



Controller

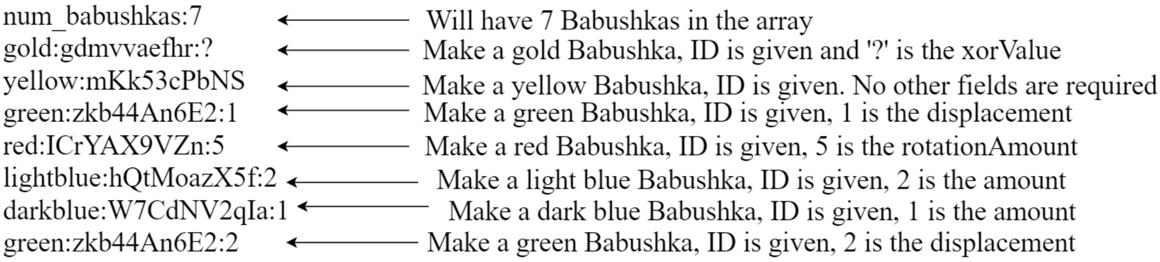
-babushkaArr : Babushka**
-numBabushkas : int

+Controller(std::string filePath)
+~Controller()
+decrypt(const unsigned char* array, int size) : ReturnStruct
+encrypt(const unsigned char* array, int size) : ReturnStruct
-expandArray(unsigned char* array, int currentSize, const unsigned char* id, int idSize) : ReturnStruct
-reduceArray(unsigned char* array, int currentSize, const unsigned char* expectedId, int idSize) : ReturnStruct
+printArray(unsigned char* array, int size) : void

Figure 9: Controller class UML diagram.

The Controller class manages the process shown in Figure 2, as well as decryption. It is responsible for appending and prepending the Babushka id to a given array (as well as removing the ids during the decryption process).

- The Controller constructor receives a file path to a text file which defines the structure of the babushkaArr. All relevant fields are separated with a colon (:). The first line provides the number of Babushkas to create, which relates to the numBabushkas member variable. The constructor must then populate the babushkaArr of type Babushka**. The specific derived classes must be placed in this array and polymorphism can then be exploited. Figure 10 explains an example text file. Top to bottom in the text file corresponds to first to last element in the babushkaArr. You may assume the text file will always be valid.

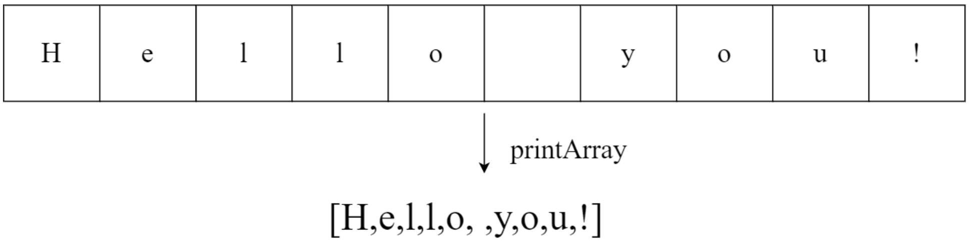


num_babushkas:7 ← Will have 7 Babushkas in the array
gold:gdmvvaefhr:? ← Make a gold Babushka, ID is given and '?' is the xorValue
yellow:mKk53cPbNS ← Make a yellow Babushka, ID is given. No other fields are required
green:zkb44An6E2:1 ← Make a green Babushka, ID is given, 1 is the displacement
red:ICrYAX9VZn:5 ← Make a red Babushka, ID is given, 5 is the rotationAmount
lightblue:hQtMoazX5f:2 ← Make a light blue Babushka, ID is given, 2 is the amount
darkblue:W7CdNV2qIa:1 ← Make a dark blue Babushka, ID is given, 1 is the amount
green:zkb44An6E2:2 ← Make a green Babushka, ID is given, 2 is the displacement

Figure 10: Example text file schema.

- The Controller destructor deallocates the babushkaArr.
- The expandArray function is responsible for appending an id to the end of a given array, as well as prepending the id to the beginning of the given array. In order to accomplish this, a new array of the correct size must be created. Elements from the old array should be copied into the correct positions in the newly allocated array as well as the id characters (similar to Figure 2). The old array should be deleted after its elements have been copied to the new array. Finally, a ReturnStruct must be returned which holds the pointer to the new array as well as the array's new size.
- The reduceArray function is responsible for removing an id from the beginning and the end of the given array. A new array of a smaller size needs to be created. Before creating this array, its size must be calculated. If the new size is negative then throw a ControllerException("size exception"). This exception can occur if the cipher text was somehow modified and made shorter. Before removing the id from the beginning and the end of the array, this function should check if the ids in the given array match the expectedId. If the ids do not match, throw a ControllerException("id mismatch exception"). A ReturnStruct should be returned which holds the new size of the array and the new array pointer. The old array should be deleted before returning.

- The `printArray` function is responsible for printing the characters within a given `unsigned char` array. The `size` of the array is provided as a parameter. This function must print each character in the array separated by a comma (except for the last element) and surrounded by square brackets. Finally, an `endl` should be printed. For example, consult Figure 11 below. You may assume that there will be at least one element in the array. Note that technically the array stores integer values, but `cout` will print the ASCII equivalents. Make sure you print in the correct format otherwise you could lose significant marks.



H	e	l	l	o		y	o	u	!

[H,e,l,l,o, ,y,o,u,!]

Figure 11: Example printing an array. Note the empty cell is actually a space character.

- The `encrypt` function is responsible for managing the encryption of a given plain text using the `Babushka`s in the `babushkaArr` similar to Figure 2. A deep copy of the provided `array` parameter needs to be made and subsequently used for encryption. The initial array should be printed using the `printArray` function. Then, for each `Babushka` in the `babushkaArr`, the following should occur:
  - The array should be expanded using `expandArr`
  - The array should be encrypted using the `encrypt` function.
  - The array should be printed using the `printArray` function.

  Finally, the resulting array should be returned as a `ReturnStruct`. The contents of this function should be surrounded by a `try-catch` block that catches a `BabushkaException` by reference. A new exception is thrown in the `catch` block of type `ControllerException` which takes the `BabushkaException` as an argument as well as the `encrypt exception` custom message string.
- The `decrypt` function is responsible for managing the decryption of a given cipher text using `Babushka`s in the `babushkaArr`. A deep copy should be made on the input `array` before decryption. This function should perform the necessary steps to undo the result of `encrypt` function. This function should have a similar `try-catch` block with the only difference being the message `decrypt exception` provided to the `ControllerException` class.

Example files

Some files have been provided. An example schema text file is given. In addition a `main.cpp` is given to show you an example of how the code can be used with the provided text file. The example output is given in the `main.cpp` as a comment in order for you to ensure your output is identical.

File check-list

You must submit the following files:

- makefile
- Babushka.h
- Babushka.cpp

- BabushkaException.h
- BlueBabushka.h
- BlueBabushka.cpp
- Controller.h
- Controller.cpp
- ControllerException.h
- ControllerException.cpp
- DarkBlueBabushka.h
- DarkBlueBabushka.cpp
- DisplacementException.h
- DisplacementException.cpp
- GoldBabushka.h
- GoldBabushka.cpp
- GreenBabushka.h
- GreenBabushka.cpp
- LightBlueBabushka.h
- LightBlueBabushka.cpp
- OverflowException.h
- OverflowException.cpp
- RedBabushka.h
- RedBabushka.cpp
- ReturnStruct.h
- RotateException.h
- RotateException.cpp
- UnderflowException.h
- UnderflowException.cpp
- YellowBabushka.h
- YellowBabushka.cpp
- main.cpp (optional)

Implementation Considerations

- You must compile using C++98 using the `-std=c++98` flag.
- There are no forward declarations in this assignment.
- Your header files may be overwritten. You may not assume that the header files use `namespace std`, but you may assume that they include the relevant libraries.

Allowed Includes

The following includes are allowed:

- cstring
- string
- fstream
- sstream
- iostream
- `X.h` where `X` is the name of a class/struct defined in this assignment specification.

Submission

You need to submit your source files on the Fitch Fork website (https://ff.cs.up.ac.za/). Place all of your `.h` files and your `.cpp` files in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number. Also place your **makefile** in this archive. There is no need to include any other files in your submission. **Do not put any folders in the .zip archive**. You have 5 submissions and your best mark will be your final mark. Do not use Fitch Fork to test your code because you have limited marking opportunities. Upload your archive to the Assignment 3 slot on the Fitch Fork website for COS110. Submit your work before the deadline. No late submissions will be accepted!