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

COS110 Assignment 3

Access Control Device

Due date: 11 October 2020, 23h00

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

In the fields of physical security and information security, access control (AC) is the selective restriction of access to a place or other resource while access management describes the process. The act of accessing may mean consuming, entering, or using. Permission to access a resource is called authorization. For this assignment, you will create a hierarchy of classic access control devices and authorization process by using hashing.

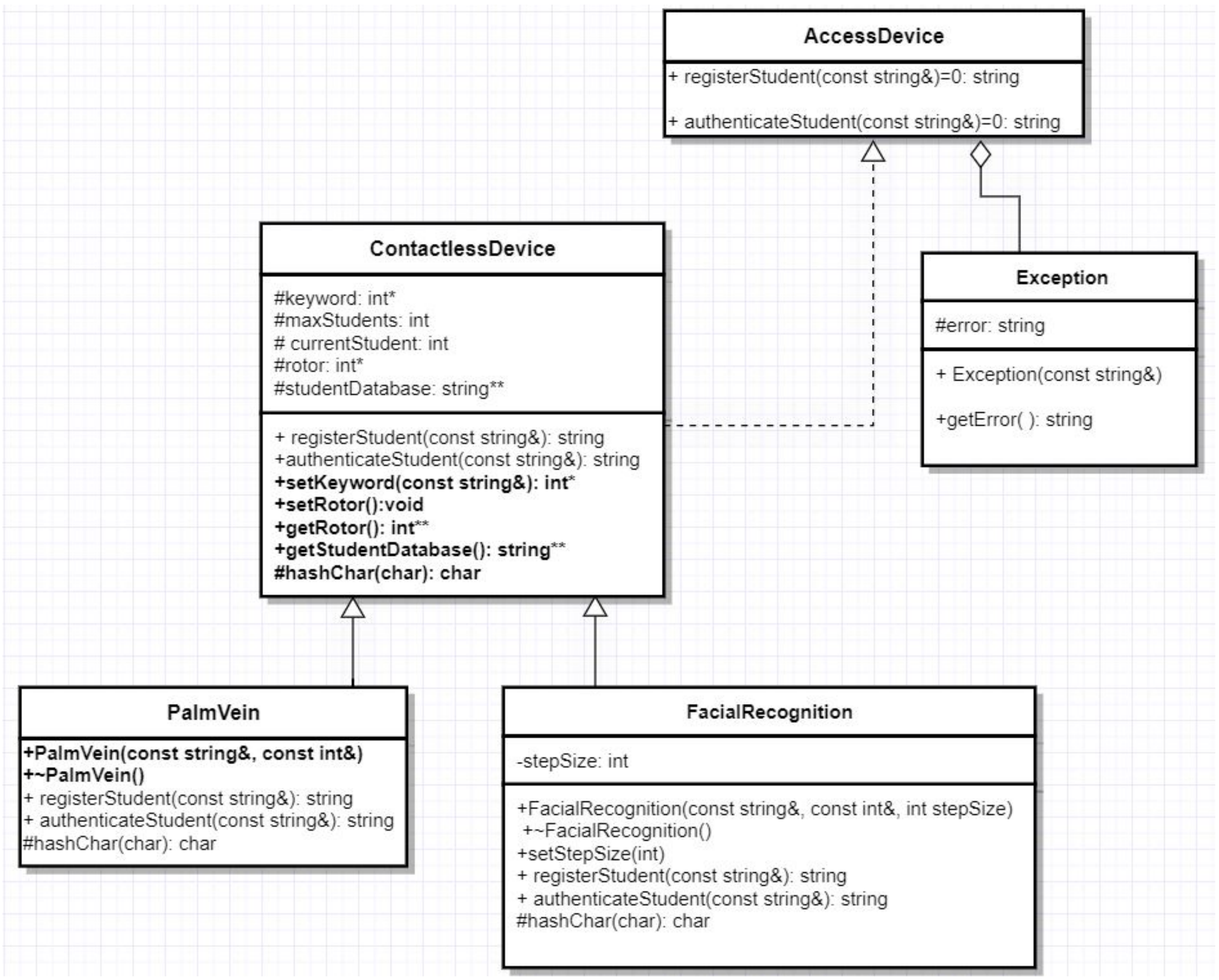
3 Your task

This assignment will give you an oppurtunity to work with inheritance, polymorphism, aggregation, exception handling, and cryptography or hashing. 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 auto marker 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. `AccessDevice, ContactlessDevice, PalmVein, Exception`
2. `FacialRecognition`

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



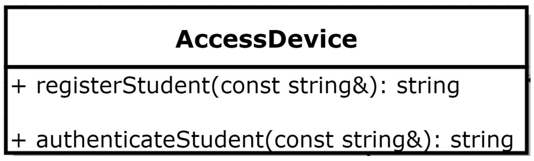
AccessDevice
+ registerStudent(const string&)=0: string
+ authenticateStudent(const string&)=0: string
ContactlessDevice
#keyword: int*
#maxStudents: int
# currentStudent: int
#rotor: int*
#studentDatabase: string**
+ registerStudent(const string&): string
+authenticateStudent(const string&): string
+setKeyword(const string&): int*
+setRotor():void
+getRotor(): int**
+getStudentDatabase(): string**
#hashChar(char): char
Exception
#error: string
+ Exception(const string&)
+getError( ): string
PalmVein
+PalmVein(const string&, const int&)
+~PalmVein()
+ registerStudent(const string&): string
+ authenticateStudent(const string&): string
#hashChar(char): char
FacialRecognition
-stepSize: int
+FacialRecognition(const string&, const int&, int stepSize)
+~FacialRecognition()
+setStepSize(int)
+ registerStudent(const string&): string
+ authenticateStudent(const string&): string
#hashChar(char): char

3.1 Task 1: Contactless access device hierarchy, and the PalmVein device

There are two major types of access devices: contact and contactless devices. For contactless devices, the user does not have to physically touch the device. The device will authanteticate the user based on the user's unique physical features like: eye, palm vein, face. For this task, you will create a hierarchy of classes representing a generic AccessDevice interface, and implement two concrete contactless access devices: PalmVein and FacialRecognition.

3.1.1 AccessDevice

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

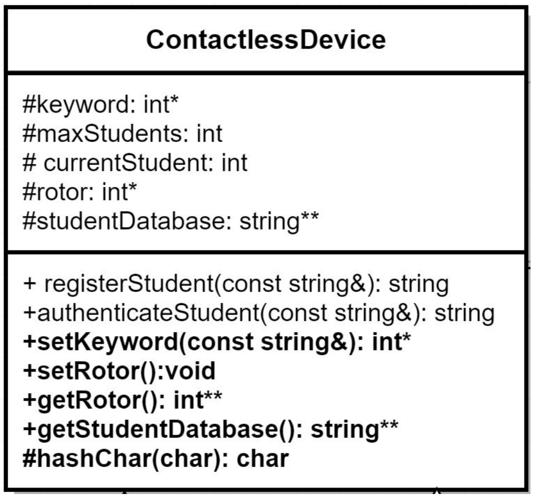

AccessDevice
+ registerStudent(const string&): string
+ authenticateStudent(const string&): string

An access control device is a device which accepts a unique input string and registers a user by adding the hash string to a database. Moreover, an access control device will authorize a user by validating against the database. Abstract `AccessDevice` declares two **pure virtual** functions:

- `string registerStudent(const string&)` – This function is abstract and must be overridden by the derived classes. The function will receive plaintext (`string`) as input, hash code it, store the hash code and the plaintext in the studentDatabase, and return the hash code (`string`).
- `string authenticateStudent(const string&)` – This function is abstract and must be overridden by the derived classes. The function will receive plaintext (`string`) as input, hash code it, and use the hash code and plaintext to check if the student exists in the `studentDatabase` .

3.1.2 ContactlessDevice

The `AccessDevice` interface can be extended to create an abstract `ContactlessDevice` class, which will become the base class for all concrete Contactless devices. Create the `ContactlessDevice` class in `ContactlessDevice.h` and `ContactlessDevice.cpp` files, according to the UML and the specification below:


ContactlessDevice
#keyword: int*
#maxStudents: int
# currentStudent: int
#rotor: int*
#studentDatabase: string**
+ registerStudent(const string&): string
+authenticateStudent(const string&): string
+setKeyword(const string&): int*
+setRotor():void
+getRotor(): int**
+getStudentDatabase(): string**
#hashChar(char): char

Characters are going to be hashed individually. For that reason one **protected** function can be added to the `ContactlessDevice` class:

- `char hashChar(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, hash it, and return a hash code character (`char`).
- `int *keyword` – the keyword stores an `n x 1` 1D array by casting each char into an int. Keywords **DO NOT** contain letters.
- `int maxStudents` – stores the maximum number of students the student database can hold.
- `int currentStudent` – stores the current number of registered students.
- `int **rotor` – rotor is used to hash a character. Explained in section 3.1.4.

- `string **studentDatabase` – stores registered students.

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

- `string registerStudent(const string&)` – This function receives plaintext (`string`) as input, hash it by applying `hashChar` function to each character of plaintext, stores plaintext and the hash code in the `studentDatabase` and returns the hash code (`string`).

- `string authenticateStudent(const string&)` – This function receives plaintext (`string`) as input, hash it by applying `hashChar` function to each character of the plaintext, and use the hash code and plaintext to check if the student exists in the `studentDatabase`. If the user exist, return a concatenation of the hash code and the Boolean result(`string`). When the user exist, return a concatenation of the hash code and the Boolean result(`string`). Eg. 2345false.

`ContactlessDevice` also implements non-abstracts functions:

- `int *setKeyword(const string&)` – This function receives plaintext (`string`) as input, and stores it in `keyword` by converting it into an `n x 1` array, and returns the stored `keyword`. This function must handle invalid input using **exceptions**. The keyword should have only numbers or integers. Throw an exception for invalid input with the message: "Invalid input", when an input character is not an integer and cannot be cast into an integer.

- `void setRotor()` – This function uses the stored keyword to set up the rotor. When a keyword is not set throw an exception with the error message: `Keyword must be set`. Section 3.1.4 give details on how to set up a rotor.

- `int **getRotor()` – This function returns the rotor. As depicted in table 1 and explained in section 3.1.4, rotors are stored and outputted as a 2D array with column one representing rotor one, column two representing rotor 2, etc.

Table 1: Rotor table. Each column corresponds to a rotor.

4	9	5
5	0	6
6	1	7
7	2	8
8	3	9
9	4	0
0	5	1
1	6	2
2	7	3
3	8	4

- `string **studentDatabase()` – This function returns the `studentDatabase`. As depicted in table 2, the database is stored and outputted with column one being the plaintext, and column two storing the corresponding hash code.

Table 2: Student Database. Left column: plaintext, right column: hash.

677899	567754
67894	09368
876375	726482

Note that the `ContactlessDevice` is still an abstract class, because it has one pure virtual functions. However, it does implement some of the functionality according to the interface of the `AccessDevice` class. The non-abstract `registerStudent` and `authenticateStudent` functions make use of the abstract `hashChar`. When a child class implements `hashChar` as prescribed by the `ContactlessDevice`, the `ContactlessDevice` will already know how to apply these functions to plaintext and hashcode. 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 Generating a hashcode

For the purpose of this assignment, we will use a lower or a simple version of the enigma machine algorithm to hash (encrypt) the student number. The Enigma machine is an encryption device developed and used in the early to mid-20th century to protect commercial, diplomatic and military communication. It was employed extensively by Nazi Germany during World War II, in all branches of the German military.

A digit in the plaintext is replaced with a digit corresponding to a certain digit up or down in the digits `[0,...,9]`. A rotor with 10 digits is rotated with a gear based on a `keyword`.

Algorithm:

- Set the number of rotors and place them in the right starting position with the help of the `keyword`.
- To hash (encrypt) each number in the number set, for each rotor, find the match from the last rotor and return the corresponding index of the matched number.

3.1.4 Setting rotor

The settings of rotor will depends on the `keyWord`. Eg. Keyword: **456**

As depicted in Table 3, there are three digits in the `keyword (456)` so we create `3 rotors`, each column representing a rotor. The number of rows is set to 10 (indices start with 0) for this assignment. The first rotor numbering starts with its corresponding keyword digit, i.e. `4`. The subsequent rotors start the numbering by summing up previous rotors starting positions with its keyword digit. If the summation is greater than `index 9`, the counting starts with `zero` again. Eg., rotor 3 is 6+9= 15, after 9, 10 becomes 0, 11 becomes 1, and so on. Thus, 15 is converted to 5.

3.1.5 PalmVein

Palm vein authentication is a vein feature authentication technology that uses palm veins as the biometric feature. Palm vein patterns are normally captured using near-infrared light via either the reflection or the transmission methods. In the reflection method, near-infrared rays are emitted towards the palm to be identified, and the reflected light is captured for authentication. As a contactless type of biometric identification, it is suitable for use in applications that require a high level of hygiene or for use in public applications, especially during a pandemic.

For the purpose of this assignment, palm vein will be represented by a plaintext of numbers.

Eg. Keyword: **456**

Plaintext number to hash and store: **7345**

According to the table 3, plaintext **7345** is converted to the following hashcode: **2890**, where 7 = index 2 of the rotor table, 3 = index 8, 4 = index 9 and 5 = index 0. Note that the last column of the rotor table is used to perform the hashing.

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

Table 3: Palm vein rotor

index	Rotor 1	Rotor 2	Rotor 3
	4	5(+4)	6(+9)
0	4	9	**5**
1	5	0	6
2	6	1	**7**
3	7	2	8
4	8	3	9
5	9	4	0
6	0	5	1
7	1	6	2
8	2	7	**3**
9	3	8	**4**

```
PalmVein
-----------------------------------------------
+PalmVein(const string&, const int&)
+~PalmVein()
+ registerStudent(const string&): string
+ authenticateStudent(const string&): string
#hashChar(char): char
```

The `PalmVein` class has the following member functions:

- `PalmVein(const string& key, const int& maxStud)` – Parameterised constructor that sets the `keyword` to the provided string (key) and the `maxStudents` to `maxStud`, which is the maximum students the database can register. Moreover, the `string **studentDatabase` is initialized and each element is set to `'z'` with the number of rows equals to `maxStudents` with two colunms.
- `~PalmVein()` – The destructor for the class. It deallocates the memory assigned to the class. It should delete `*keyword`, `int **rotor` and `string **studentDatabase` from the first index.
- `char hashChar(char)` – this function implements `PalmVein` hashing. A plaintext `char` is received as input and shifted according to the rotor. The index of the rotated `char` is returned. No error-checking has to be done in this function.
- Note that `registerStudent(string)` and `authenticateStudent(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.

  Additionally, create the following `Exception` class:

```
Exception
---------------------------
#error: string
---------------------------
+Exception(const string&)
+getError(): string
```

  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 `keyword` should only be accepted if it is more than one character long, and if it is not made entirely of ' ' (space) characters. If the keyword is either too short, or made out of spaces, throw an `Exception` object initialised with the following message: “The keyword provided is not going to generate a safe encryption”. The exception message should contain **no punctuation** and **no new line characters**. This applies to both the constructor and `setKeyword` fucntion.

Throw an exception with the message: "Student Database is full", when registering more than the `maxStudents`.

Throw an exception with the message: "Student already exist", if the student is already registered.

`PalmVein` class is not an abstract class, therefore you can instantiate a `PalmVein` object and test your code. Test your encoding thoroughly before submitting for marking.

3.1.6 Test and submit for marking - 60 marks

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. The following classes should be in the folder: `PalmVein.h/cpp`, `ContactlessDevice.h/cpp`, `Exception.h`, `AccessDevice.h`.

3.2 Task 2: FacialRecognition

`PalmVein` hashing is simple and static, and therefore not very secure, especially if a short and simple `keyword` is used. You are going to create a much stronger and dynamic hash code for `FacialRecognition`. In `FacialRecognition` hash code, the rotor is reconfigured after each character is hashed based on the `stepSize`.

For this assignment’s approach to `FacialRecognition` hashing, you will hash a character as follows:

Every character of plaintext will be encoded similarly to the `PalmVein`, but the starting position of each rotor will change after each character based on `stepsize`.

Eg. Keyword: **456**, stepSize : **2**

Plaintext number to hash and store: **7345**

Since there are four characters in the plaintext to hash, the process will go through 4 iterations. Each Iteration will reconfigure the rotor to a new starting positions a depicted in tables 4 to 7. The first rotor will gear down by 2 steps (`stepSize`) setting the starting position to 6 (4+2) after the first hash or iteration. The first rotor will again be set to 8 (6+2) after the second iteration. This process will continue for all characters.

Take note of the following iteration process:

- Iteration 1: Uses the default keywords and rotors.
- Iteration 2: Uses the default keywords. However, the keyword of rotor 1 or the starting position of rotor is geared down by `stepSize`.
- Iteration 3 to `n` iterations: the keywords are set to the starting positions of iteration `n - 1`.

Hashcode: **2680**.

where 7 = index 2 in iteration 1, 3 = index 6 in iteration 2, 4 = index 8 in iteration 3 and 5 = index 0 in iteration 4.

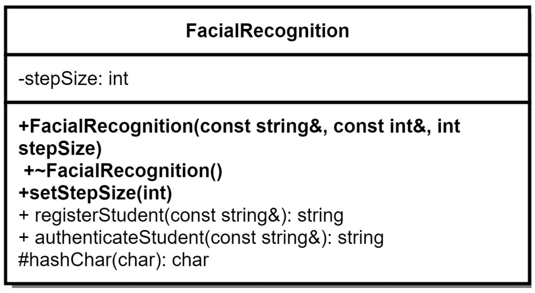
Table 4: Iteration 1

index	Rotor 1	Rotor 2	Rotor 3
	4	5(+4)	6(+9)
0	4	9	5
1	5	0	6
2	6	1	**7**
3	7	2	8
4	8	3	9
5	9	4	0
6	0	5	1
7	1	6	2
8	2	7	3
9	3	8	4

Table 5: Iteration 2

index	Rotor 1	Rotor 2	Rotor 3
	4(+2)	5(+6)	6(+1)
0	6	1	7
1	7	2	8
2	8	3	9
3	9	4	0
4	0	5	1
5	1	6	2
6	2	7	**3**
7	3	8	4
8	4	9	5
9	5	0	6

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



FacialRecognition

-stepSize: int

+FacialRecognition(const string&, const int&, int stepSize)
+~FacialRecognition()
+setStepSize(int)
+ registerStudent(const string&): string
+ authenticateStudent(const string&): string
#hashChar(char): char

The stepSize is stored in a private member variable:

- `int stepSize` – The `stepSize` value used to determine how the rotor is geared.

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

- `FacialRecognition(const string& key, const int& maxStudents, int stepSize)` – Parameterised constructor that sets the `keyword` to the provided string and the maximum students the database can register. Moreover, the `string **studenDatabase` is initialized and set to `'z'`, and sets the `stepSize` value to the specified `stepSize`.

- `~FacialRecognition()` – The destructor for the class. It deallocates the memory assigned to the class. It should delete `*keyword`, `int **rotor` and `string **studentDatabase` from the first index.

Table 6: Iteration 3

index	Rotor 1 6(+2)	Rotor 2 1(+8)	Rotor 3 7(+9)
0	8	9	6
1	9	0	7
2	0	1	8
3	1	2	9
4	2	3	0
5	3	4	1
6	4	5	2
7	5	6	3
8	6	7	**4**
9	7	8	5

Table 7: Iteration 4

index	Rotor 1 8(+2)	Rotor 2 9(+0)	Rotor 3 6(+9)
0	0	9	**5**
1	1	0	6
2	2	1	7
3	3	2	8
4	4	3	9
5	5	4	0
6	6	5	1
7	7	6	2
8	8	7	3
9	9	8	4

- `void setStepSize(int)` – Sets the stepSize value to the specified stepSize. 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 `hashChar()` function must be overidden in the `FacialRecognition` to implement hashing 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.

`FacialRecognition` class is not an abstract class, therefore you can instantiate a `FacialRecognition` object and test your code.

3.2.1 Test and submit for marking - 40 marks

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. The following classes should be in the folder: `FacialRecognition.h/cpp`, `ContactlessDevice.h/cpp`, `Exception.h`, `AccessDevice.h`. Also note that the number of uploads is limited and should be used wisely.

You will be allowed to use the following libraries for each class:

- iostream
- string

- sstream

`The End`