Rapport de laboratoire

**HEIG-VD**

TIC

Laboratoire POA

Labo 1: Matrix Reloaded

**Réalisé par :**

Ivan Vecerina

**A l'attention de :**

M. Donini

M. Decorvet

**Dates :**

Début du laboratoire : 2 février 2022

Fin du laboratoire : 17 mars 2022

Table des matières

[1 Introduction 3](#_Toc98381614)

[2 Choix de modélisation 3](#_Toc98381615)

[2.1 MyRandom 3](#_Toc98381616)

[2.2 Operation 3](#_Toc98381617)

[2.2.1 Add 3](#_Toc98381618)

[2.2.2 Sub 3](#_Toc98381619)

[2.2.3 Multiply 3](#_Toc98381620)

[2.3 Matrix 3](#_Toc98381621)

[2.3.1 Allocate() et Deallocate() 3](#_Toc98381622)

[2.3.2 Constructeurs 4](#_Toc98381623)

[2.3.3 Getters 4](#_Toc98381624)

[2.3.4 Opérateurs 4](#_Toc98381625)

[2.3.5 swapWith() 4](#_Toc98381626)

[2.3.6 doOperation() 4](#_Toc98381627)

[2.3.7 Opérations inplace 4](#_Toc98381628)

[2.3.8 Opérations retournant une Matrice (copie) 4](#_Toc98381629)

[2.3.9 Opérations retournant un pointeur Matrice (copie) 4](#_Toc98381630)

[3 Tests 5](#_Toc98381631)

# Introduction

Dans ce laboratoire, nous devons implémenter un classe matrice en c++ tout comme nous l’avions fait en Java pour POO (labo 5). Tout comme en POO, le constructeur devait prendre les dimensions de la matrice ainsi que le modulo qui limitera nos valeurs. Nous créons une classe abstraite “Operation” ainsi que ses classes filles : Add, Sub, Multiply (respectivement somme, soustraction, multiplication).

# Choix de modélisation

## MyRandom

La classe *MyRandom* est implémentée de manière analogue à un singleton afin d’éviter de définir plus d’une fois le “seed” car cela gâchera la pseudo-randomisation dont on fait usage. La méthode “getRandMod” contient une variable statique de type MyRandom faisant appel au constructeur par défaut que la première fois (par le statique) qui initialisera le seed.

## Operation

Cette classe est abstraite. Elle met en place la fonction floorMod permettant de faire des modulos de nombres négatifs de manière analogue à celle de Math.floorMod() du Java. Elle met aussi en place la méthode abstraite operation() qui applique une opération binaire selon la définition dans ses classes filles.

### Add

Cette classe permet hérite de Operation et défini la méthode operation() pour l’addition et retournant un résultat normalisé par floorMod().

### Sub

Cette classe permet hérite de Operation et défini la méthode operation() pour la soustraction et retournant un résultat normalisé par floorMod().

### Multiply

Cette classe permet hérite de Operation et défini la méthode operation() pour la multiplication et retournant un résultat normalisé par floorMod().

## Matrix

Cette classe est la classe principale à implémenter et également celle sur laquelle nous avons le plus de libertés. Ayant regardé le cahier des charges plusieurs idées nous sont venues en tête et nous avons été mener à prendre plusieurs décisions vis-à-vis de l’implémentation.

### Allocate() et Deallocate()

La première décision majeure fut de faire de l’allocation et la désallocation dynamique de la mémoire, ayant été forces d’utiliser tableau à la C, des méthodes à par entières. Cette décision nous permet de préciser lors de l’allocation au travers de paramètres avec valeurs par défaut si l’on veut juste faire l’allocation, la faire avec un remplissage de valeurs aléatoires ou encore de copier les valeurs d’une autre matrice.

Cette approche nous permet de factoriser le code afin d’éviter d’utiliser plusieurs méthodes similaires à la place. Afin d’éviter d’embrouiller l’utilisateur, la fonction d’allocation prends des paramètres par défaut afin de lui offrir une meilleure expérience.

### Constructeurs

Les modulos négatifs étant réelles, l’approche que l’on a prise est d’accepter un int comme paramètre du modulo et de retourner une runtime\_error afin de ne pas induire l’utilisateur en erreur el lui laissant croire que le modulo négatif est accepté alors qu’il ne sera qu’implicitement casté.

Le constructeur par copie est implémenté de manière relativement standard.

Cependant, en plus des deux constructeurs ci-dessus, nous en avons également créer un troisième prenant deux matrices par références constantes en entrée qui nous construit une matrice avec les dimensions les plus grandes des deux matrices en paramètres. C’est aussi dans celui-ci que l’on vérifie que les modulos des deux matrices correspondent bien pour les opérations.

### Getters

Getters pour la hauteur, largeur et un élément à une ligne x colonne y ont été implémentés pour l’opérateur de flux.

### Opérateurs

Pour ce labo nous devons surcharger l’opérateur de flux (en fonction amie) ainsi que l’opérateur d’affectation. Les deux opérateurs sont implémentés comme demandé sans trop de prise de décision de notre part.

### swapWith()

Cette méthode est un swap pour matrices qui swap tout leurs attributs.

### doOperation()

Cette méthode statique est la méthode clé de ce laboratoire. Elle prend en paramètre deux matrices par références constantes (les opérandes), une matrice par référence dans laquelle on stockera le résultat de l’opération des deux opérandes, l’opération de type Operation étant également passé par référence constante.

Cette méthode nous a permis de factoriser les 9 méthodes d’opérations malgré leur type de retour étant différents.

### Opérations inplace

La logique de ces opérations est simple. Création d’une matrice temporaire utilisant notre 3ème constructeur, ainsi que déclaration de l’Operation. Suivi de notre doOperation() avant de finir sur un swap the tmp et this. Tmp est supprimée en fin de fonction donc mémoire est OK.

### Opérations retournant une Matrice (copie)

La logique de ces opérations est similaire aux premières. Création d’une matrice résultat utilisant notre 3ème constructeur, ainsi que déclaration de l’Operation. Suivi de notre doOperation() avant de finir sur un retour d’un copie de la matrice résultat (se fait implicitement) tmp et this. La matrice résultat de la fonction étant supprimée en fin de fonction donc mémoire est OK.

### Opérations retournant un pointeur Matrice (copie)

La logique de ces trois dernières opérations est un peu plus délicate. Devant retourner un pointeur sur Matrix, nous devons créer la matrice résultat avec un **new** utilisant notre 3ème constructeur, afin d’éviter la suppression de l’objet pointé en fin de fonction,

Cependant, cela implique que l’on transfert l’ownership de l’objet à l’utilisateur. Il faut donc s’assurer de faire un **delete** de l’objet pointé après son utilisation.

Cette approche n’est pas idéale mais les contraintes données rendent difficile de faire autrement avec des pointeurs classiques.

# Tests

|  |  |  |
| --- | --- | --- |
| Action effectuée | Résultat attendu | Résultat obtenu juste ? |
| **Création de matrices** | | |
| Matrice 4x4 de modulo 1 | Matrice 4x4 remplie de 0 | OUI |
| Matrice 0x0 de modulo 5 | Matrice vide | OUI |
| Matrice 1x4 de modulo 5 | Matrice ligne | OUI |
| Matrice 4x1 de modulo 5 | Matrice colonne | OUI |
| Matrices 3x4 et 4x3 de modulo 5 | Matrice génériques 3x4 et 4x3 | OUI |
| Matrice par copie de matrice générique | La même matrice | OUI |
| Matrice par affectation de matrice générique | La même matrice | OUI |
| **Opérations (vérifié avec debug + sur papier)** | | |
| Matrice affectée à elle même | Inchangé | OUI |
| Operations matrice vide avec matrice colonne | Résultat correcte ?  Sur le 3 opérations, les 3 retours correspondes ? | OUI |
| Operations matrice vide avec matrice ligne | Résultat correcte ?  Sur le 3 opérations, les 3 retours correspondes ? | OUI |
| Operations matrice ligne avec matrice colonne | Résultat correcte ?  Sur le 3 opérations, les 3 retours correspondes ? | OUI |
| Operations matrice générique avec matrice générique | Résultat correcte ?  Sur le 3 opérations, les 3 retours correspondes ? | OUI |
| **Erreurs** | | |
| Matrice modulo = 0 | Runtime\_error | OUI |
| Matrice modulo < 0 | Runtime\_error | OUI |
| Addition inplace entre modules différents | Invalid\_argument | OUI |
| Soustraction inplace entre modules différents | Invalid\_argument | OUI |
| Multiplication inplace entre modules différents | Invalid\_argument | OUI |
| Addition retournant Matrix entre modules différents | Invalid\_argument | OUI |
| Soustraction retournant Matrix entre modules différents | Invalid\_argument | OUI |
| Multiplication retournant Matrix entre modules différents | Invalid\_argument | OUI |
| Addition retournant Matrix\* entre modules différents | Invalid\_argument | OUI |
| Soustraction retournant Matrix\* entre modules différents | Invalid\_argument | OUI |
| Multiplication retournant Matrix\* entre modules différents | Invalid\_argument | OUI |