Matrice – Labo 1

Durée du travail :	**Du 29.02.2024 au 20.03.2024**
Auteurs :	**Demont Killian** **Graf Calvin**
Enseignant :	**Krähenbühl Grégoire**
Assistant :	**Decorvet Grégoire**
Domaine d'application :	**C++**
Lieu de travail :	**HEIG-VD | Yverdon-les-Bains**

Table des matières

1. Introduction

Ce laboratoire a pour objectif de concevoir une classe « Matrix » capable de représenter des matrices de taille variable contenant des entiers entre 0 et n-1. Avec n étant un entier positif spécifié par l'utilisateur et donc cela implique que les valeurs des matrices sont modulo n.

De plus, il faut pouvoir afficher le contenu d'une matrice avec l'opérateur d'écriture dans un flux ainsi que les opérations de construction et d'assignement par duplication et par déplacement.

La classe doit permettre d'effectuer des opérations, composante par composante entre deux matrices telles que l'addition, la soustraction et la multiplication. Toutes ces opérations devront être réalisées modulo n.

Enfin, nous devrons implémenter la gestion d'exceptions en cas d'erreur, par exemple si l'on souhaite créer une matrice avec un nombre de lignes négatif ou encore si l'on essaie de créer une matrice avec des valeurs invalides.

2. Développement

2.1 Instruction de compilation

Nom du compilateur : Mingw-w64 gcc

Version du compilateur : 13.2.0

2.2 Choix d'implémentation

Nous avons factorisé les opérations mathématiques afin de faciliter l'ajout de nouvelles opérations ultérieurement.

Puisque ce n'était pas demander explicitement dans le laboratoire et après la discussion que nous avons eu (Calvin Graf), nous avons décidé de ne pas ajouter la gestion de l'allocation dynamique avec std::bad_alloc afin de ne pas nous complexifier la tâche.

Manières d'implémentation	Avantages	Désavantages
En modifiant la matrice sur laquelle est invoquée la méthode (opérateurs +=, -= et *=)	• Économie de mémoire : pas besoin d'allouer de la mémoire supplémentaire pour stocker le résultat	• La modification est destructive • Si la matrice est partagée entre plusieurs parties du code, la modification peut entraîner des effets secondaires imprévus.
En retournant, par valeur une nouvelle matrice résultat allouée statiquement (opérateurs +, - et *)	• Simple à utiliser car elle retourne directement le résultat sous forme d'une nouvelle instance de matrice. • La matrice d'origine reste inchangée.	• Opérations qui peuvent être coûteuses en termes de performances et de mémoire, surtout pour les matrices de grande taille.
En retournant, un pointeur sur une nouvelle matrice résultat allouée dynamiquement.	• Économie de mémoire : le résultat est stocké dans une nouvelle matrice allouée dynamiquement, ce qui évite les opérations coûteuses de copie. • Flexibilité : l'utilisateur peut décider de conserver ou de supprimer la matrice résultante en fonction de ses besoins.	• Le programmeur doit prendre des précautions supplémentaires pour gérer la mémoire correctement afin d'éviter les fuites de mémoire. • L'utilisation de pointeurs peut rendre le code plus complexe et plus sujet aux erreurs.

Dans le cas des opérateurs +, - et *, il est un important de retourner une matrice par valeur et non par référence car si nous renvoyons une référence cette dernière pointerais sur une matrice qui n'existe plus.

2.3 Protocole de tests

Test	Résultat
Construction d'une matrice avec valeur aléatoire	OK
Constructeur par copie	OK
Constructeur par déplacement	OK
Assignation par copie	OK
Assignation par déplacement	OK
Opérateur de flux	OK
Méthode statique qui retourne par valeur l'addition de deux matrices	OK
Méthode dynamique qui retourne un pointeur sur l'addition de deux matrices	OK
Méthode qui additionne la matrice en paramètre à la matrice qui l'invoque	OK
Méthode statique qui retourne par valeur la soustraction de deux matrices	OK
Méthode dynamique qui retourne un pointeur sur la soustraction de deux matrices	OK
Méthode qui soustrait la matrice en paramètre à la matrice qui l'invoque	OK
Méthode statique qui retourne par valeur la multiplication de deux matrices	OK
Méthode dynamique qui retourne un pointeur sur la multiplication de deux matrices	OK
Méthode qui multiplie la matrice en paramètre à la matrice qui l'invoque	OK
Possible de créer une matrice de 0x0	OK
Si taille de la matrice négatif (ligne ou colonne), retourne runtime_error	OK
Si taille du modulo négatif, retourne runtim_error	OK
Si lors d'une opération, les deux modulos ne sont pas identique, retourne invalid_argument	OK
Accepter uniquement des arguments entre 0 (ou 1 pour le modulo) et 2147483647	OK
Si on fait une opération entre une matrice M1 × N1 et une matrice M2 × N2 et que les tailles ne correspondent pas, le résultat est une matrice max(M1, M2) × max(N1, N2) où les opérandes manquants sont remplacés par des 0.	OK

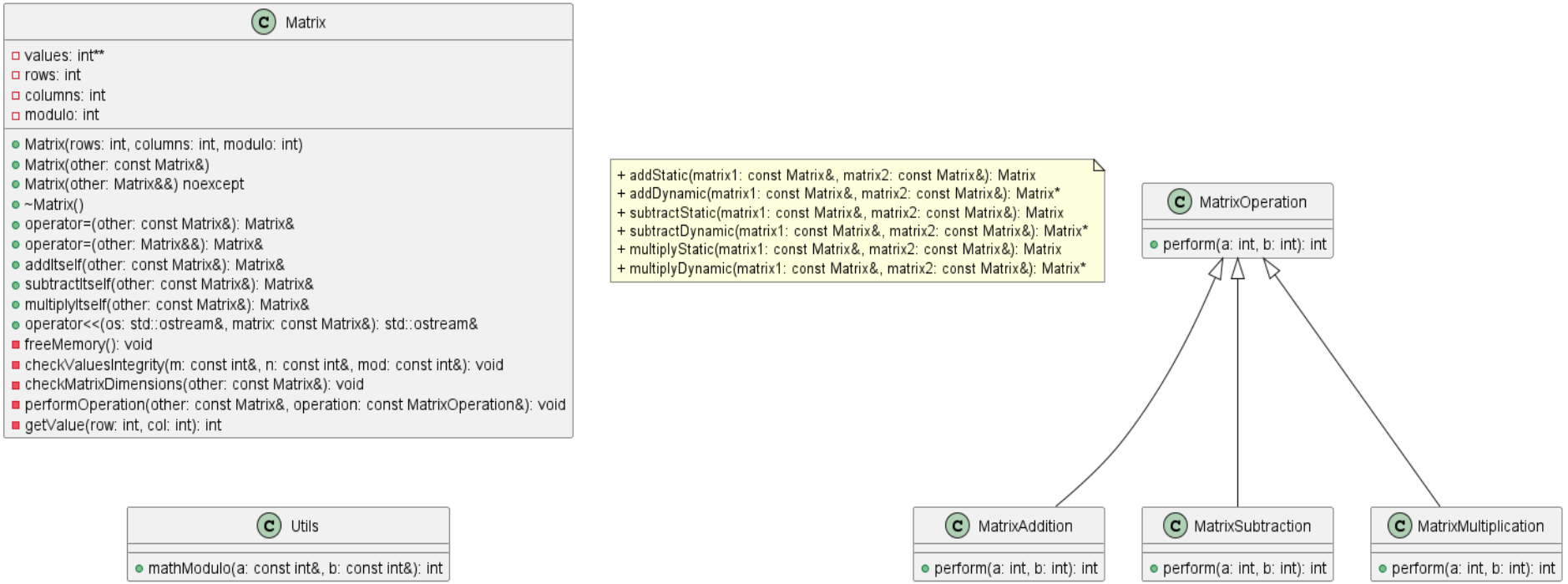
Précision : Nous acceptons 2147483647 comme argument, cependant nous n'avons pas ajouté la gestion des exceptions lorsqu'une opération qui dépassera la valeur maximum de int sera dépassé ou qu'un bad_alloc aura lieu car trop grand.

3. Conclusion

La classe Matrix permet la génération, la manipulation et l'affichage de matrice de différentes tailles tout en ayant une gestion des erreurs en particulier dans les cas limites.

La conception des classes opérations a été implémenté en factorisant le plus possible afin de faciliter l'implémentation d'éventuelles nouvelles opérations dans le futur. La règle des 5 a été mis en place ainsi que l'opérateur d'écriture dans le flux. Tous les points du cahier des charges ont été remplis et testés avec succès.

4. UML


Matrix
values: int**
rows: int
columns: int
modulo: int
Matrix(rows: int, columns: int, modulo: int)
Matrix(other: const Matrix&)
Matrix(other: Matrix&&) noexcept
~Matrix()
operator=(other: const Matrix&): Matrix&
operator=(other: Matrix&&): Matrix&
addItself(other: const Matrix&): Matrix&
subtractItself(other: const Matrix&): Matrix&
multiplyItself(other: const Matrix&): Matrix&
operator<<(os: std::ostream&, matrix: const Matrix&): std::ostream&
freeMemory(): void
checkValuesIntegrity(m: const int&, n: const int&, mod: const int&): void
checkMatrixDimensions(other: const Matrix&): void
performOperation(other: const Matrix&, operation: const MatrixOperation&): void
getValue(row: int, col: int): int
+ addStatic(matrix1: const Matrix&, matrix2: const Matrix&): Matrix
+ addDynamic(matrix1: const Matrix&, matrix2: const Matrix&): Matrix*
+ subtractStatic(matrix1: const Matrix&, matrix2: const Matrix&): Matrix
+ subtractDynamic(matrix1: const Matrix&, matrix2: const Matrix&): Matrix*
+ multiplyStatic(matrix1: const Matrix&, matrix2: const Matrix&): Matrix
+ multiplyDynamic(matrix1: const Matrix&, matrix2: const Matrix&): Matrix*
MatrixOperation
perform(a: int, b: int): int
Utils
mathModulo(a: const int&, b: const int&): int
MatrixAddition
perform(a: int, b: int): int
MatrixSubtraction
perform(a: int, b: int): int
MatrixMultiplication
perform(a: int, b: int): int