HEIG-VD — POA

Laboratoire 1 – Rapport

[redacted]

20 mars 2024

Ce document décrit nos choix et hypothèses de travail effectués pour réaliser le laboratoire n° 1 dans lequel nous avons dû créer une classe permettant la modélisation de matrices de taille quelconque. La classe doit en outre permettre de gérer des opérations arithmétiques effectuées composante-par-composante, en trois formats différents.

1 Modélisation

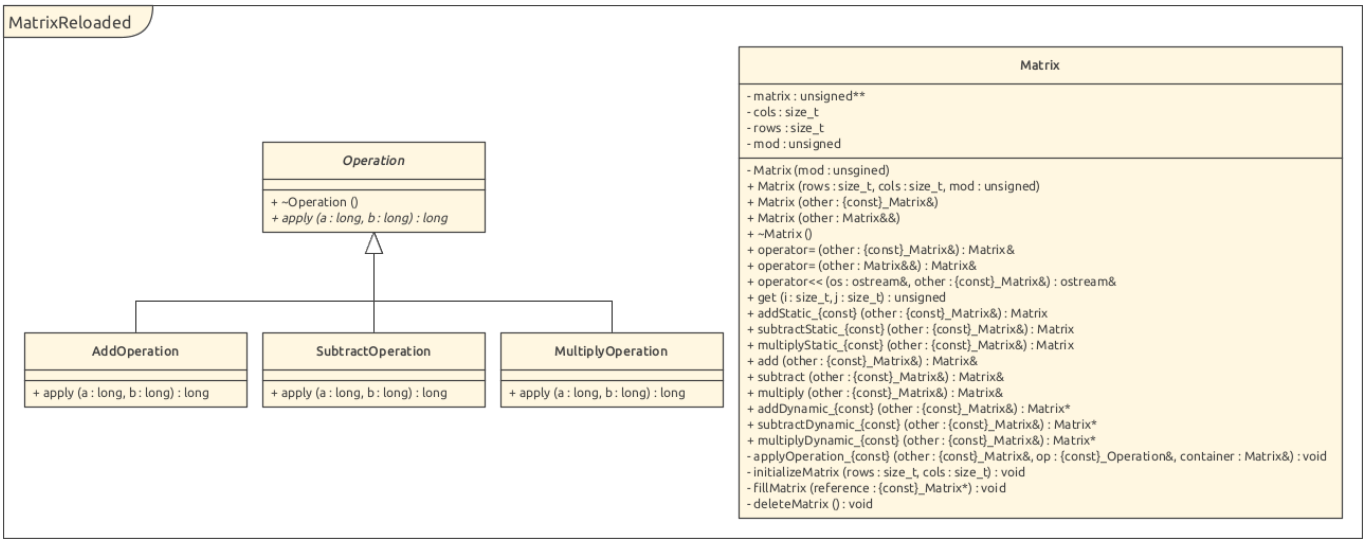

FIGURE 1 – Diagramme UML de la conception de l'implémentation

2 Instructions de compilation

Nous avons conservé la structure avec *cmake* pour notre implémentation.

Un exécutable additionnel a été défini pour nos tests et qui nécessite la librairie de tests unitaires googletest. Nous avons utilisé la version « live at head », c'est-à-dire la révision n° 110 de la `1.14.0` au commit `eff443c6` pour compiler et exécuter nos tests. Il faut toutefois noter que ce système est optionnel et que si la librairie n'est pas trouvée par *cmake* l'exécutable des tests ne sera pas généré.

Nos versions de la suite de compilation utilisée par *cmake* sont les suivantes :

- **cmake** : 3.28.3
- **ninja** : 1.11.2
- **gcc** : 13.2.1 20230801
- Compilé sur Linux avec le kernel 6.7.10

3 Choix et hypothèses de travail

3.1 Classe de matrice

- La classe `Matrix` représente la matrice à l'aide d'un double pointeur de type `unsigned` qui est alloué à la création. Le nombre de lignes et de colonnes est sauvegardé comme attribut.
- Le modulo doit être plus grand ou égal à 1. On accepte la création d'une matrice vide contenant 0 lignes et 0 colonnes.

- Nous avons décidé d'utiliser des `unsigned` pour les éléments de la matrice ainsi que le modulo, car nous ne devrions pas avoir d'éléments négatifs
- Nous avons implémenté deux constructeurs. Un constructeur public prenant en paramètre le nombre de lignes, colonnes, le modulo et créant une matrice avec des valeurs aléatoires, ainsi qu'un autre constructeur privé, prenant seulement le modulo qui crée une matrice vide. Ce dernier sera utilisé lors de l'implémentation des opérations pour éviter d'allouer inutilement les tableaux contenant les éléments.
- Comme notre classe gère un tableau à deux dimensions, la règle des trois a été implémentée au moyen d'un constructeur de copie public, un opérateur d'assignation et un destructeur. Nous avons également défini les opérations de construction par déplacement définies par la règle des cinq, ce qui nous permet en outre de simplifier la gestion des opérations.
- La classe fournit un getter qui permet de récupérer l'élément à l'indice demandé. Il retourne l'élément si les indices sont dans les dimensions de la matrice ou 0 sinon.

3.2 Opérations

- On définit une classe `Operation` servant d'interface contenant une méthode `apply` qui permettra d'appliquer une opération entre deux éléments de type `long`.
- On définit ensuite comme sous-classes de `Operation`, les classes `AddOperation`, `SubtractOperation` et `MultiplyOperation` qui définissent respectivement l'addition, la soustraction et la multiplication.
- La classe `Operation` redéfinit le destructeur par défaut en le marquant virtuel, pour permettre à une spécialisation d'éventuellement définir un destructeur propre à son implémentation.

3.3 Trois manières d'implémenter les opérations

Il était demandé d'implémenter les trois opérations de trois manières différentes, tout en conservant une bonne factorisation du code minimisant le nombre d'allocations effectuées.

Implémentation

- Une méthode privée principale `applyOperation` est définie dans `Matrix` qui sera utilisée pour les trois types d'opérations. Un appel à cette fonction doit initialiser une matrice sans allouer le tableau en mémoire (au moyen de notre constructeur privé prévu à cet effet) qui va servir d'enveloppe où le résultat sera stocké.
- Afin de permettre uniformément tous les appels à la méthode, entre autres si la matrice résultat est la même que la première matrice de l'opération, l'enveloppe vide sera toujours utilisée et initialisée par la méthode `applyOperation` à la taille du résultat dans laquelle nous mettons successivement les résultats des opérations.
- Dans le cas des opérations statiques, la méthode spécifique à l'opération peut simplement initialiser l'enveloppe vide et la retourner directement.
- Pour les opérations par allocation dynamique et de modification directe, nous utilisons la sémantique de déplacement en faisant appel à la méthode équivalente retournant une instance par valeur. L'allocation dynamique utilisera donc la r-value retournée par sa contrepartie statique dans un constructeur par déplacement, ce qui évite toute allocation supplémentaire et inutile des données. Les méthodes de modification directes utiliseront, elles, l'assignation par déplacement sur l'instance courante, permettant d'instancier le tableau résultat à la taille correcte.
- Les méthodes qui assignent le résultat directement à la matrice de gauche retournent une référence vers l'instance de la classe, permettant éventuellement à un appelant de chaîner plusieurs appels consécutifs.
- Un désavantage très mineur de cette approche est que dans le cas de la modification directe de la matrice, une nouvelle matrice serait instanciée même si la matrice existante est déjà suffisante. Nous pensons toutefois que cette approche est suffisamment optimale pour la plupart des cas d'utilisation et que son implémentation requérait une approche qui empêcherait la factorisation actuelle, alourdissant le code pour un bénéfice minime.

3.4 Avantages et désavantages

On décrit ci-dessous les avantages et désavantages des différentes méthodes implémentées.

3.4.1 En modifiant la matrice sur laquelle est invoquée la méthode

Cette méthode consiste à modifier directement la matrice sur laquelle est invoquée l'opération.

Avantages

- **Chaînage d'opérations** : Permet un enchaînement fluide des opérations sur une matrice, facilitant l'écriture de code compact et expressif

Désavantages

- **Copies inutiles** : Une erreur courante avec l'approche utilisée serait de stocker le résultat donné dans une variable non-référence de type `Matrix` (au lieu de `Matrix&`), ce qui causera automatiquement une copie probablement non désirable.

3.4.2 En retournant une matrice allouée dynamiquement

Cette méthode retourne un pointeur vers une nouvelle matrice allouée dynamiquement sur le tas. L'utilisateur est responsable de la libération de cette mémoire une fois qu'elle n'est plus nécessaire.

Avantages

- **Flexibilité** : Permet de travailler avec des matrices de grande taille sans être limité par la taille de la pile.
- **Préservation des originaux** : Les matrices originales ne sont pas modifiées, ce qui permet de les réutiliser pour d'autres opérations

Désavantages

- **Gestion de la mémoire** : Nécessite une gestion explicite de la mémoire par le programmeur, augmentant la complexité du code et le risque de fuites mémoire.
- **Performance** : L'allocation et la désallocation dynamiques sont généralement plus coûteuses en termes de performance que l'utilisation de la mémoire statique.

3.4.3 En retournant une matrice allouée statiquement

Dans cette version, la matrice est retournée par valeur. Il est évidemment impossible de retourner une référence sur une variable qui est initialisée localement à la méthode de l'opération, car elle pourrait être détruite avant que l'appelant ne puisse l'utiliser.

Avantages

- **Simplicité** : Plus simple à utiliser que l'allocation dynamique, car il n'est pas nécessaire de la désallouer explicitement.
- **Sécurité** : Pas de risque de fuites mémoire puisque la mémoire statique sera automatiquement nettoyée lorsque l'appelant terminera son traitement.

Désavantages

- **Limitation de taille** : La taille de la pile est limitée, ce qui peut être un problème pour des matrices très grandes.
- **Coût de copie** : Retourner une matrice par valeur implique de copier l'objet, ce qui peut être coûteux en termes de performances. Cela dépend cependant beaucoup du compilateur et de ses paramètres utilisés ; dans la plupart des cas, la présence du constructeur de déplacement et de l'assignation par déplacement devrait permettre à un utilisateur de cette méthode de stocker le résultat dans une variable sans nécessairement causer une copie.

4 Tests effectués

Le programme appelant demandé a été utilisé pour tester quelques effets de bords et s'assurer que les paramètres soient bien vérifiés avant de les passer plus loin. Nous avons également implémenter des tests unitaires au moyen de la librairie googletest, dont le détail sommaire est le suivant :

Scénario	Résultat attendu
Appeler un opération sur deux matrices de modulo différent	Une exception de type `std::invalid_argument` est levée
Construire une matrice avec un modulo égal à 0	Une exception de type `std::invalid_argument` est levée
Afficher une matrice vide	L'opération fonctionne normalement et affiche `[]`
Afficher une matrice	L'opération fonctionne normalement et affiche les valeurs générées aléatoirement
Ajouter, soustraire et multiplier des matrices en prenant le résultat par valeur	L'opération fonctionne, les matrices sources ne sont pas modifiées et le résultat est correct
Ajouter, soustraire et multiplier des matrices en prenant le résultat par pointeur	L'opération fonctionne, les matrices sources ne sont pas modifiées et le résultat est correct
Ajouter, soustraire et multiplier des matrices en écrasant la matrice gauche	L'opération fonctionne, la matrice droite n'est pas modifiée et l'instance retournée est bien identique à la matrice gauche modifiée
Ajouter, soustraire et multiplier des matrices avec au moins une matrice vide	L'opération fonctionne avec les valeurs absentes substituées par des zéros
Fonctionnement de la méthode utilitaire `floorMod`	Fonctionne avec des valeurs négatives, nulles ou positives

Table 1 – Tests effectués avec la classe `Matrix`