Programmation orientée objet en C++

Travaux pratiques séance 3

A la fin du TP, déposez une archive avec votre travail (code source) sur ecampus, dans le repertoire correspondant à votre groupe. Avant minuit, déposez également sur ecampus le compte-rendu détaillé (CR) correspondant à votre travail ; les modifications dans le code ultérieures à la fin du TP ne sont pas prises en compte - concentrez vous sur la rédaction du CR.
Conseils de base :

- le TP est **individuel** ; toute fraude sera sanctionnée drastiquement.
- dès que possible (toutes les quelques lignes idéalement), **compilez** et **testez** votre code. L'erreur la plus fréquente des débutants est d'écrire des dizaines de lignes de code sans tester, et de se retrouver tout à coup avec plein d'erreurs de compilation et de bugs
- si vous dupliquez du code dans vos méthodes, cela veut dire qu'il y a probablement un problème : soit une méthode devrait appeler l'autre, soit le code commun devrait être mis dans une méthode auxiliaire, qui est appelée chaque fois que sa fonctionnalité est nécessaire
- même si cela ne vous rapporte pas une meilleure note tout de suite, essayez d'aller au bout de ce sujet avant le TP suivant. Cela vous améliorera sûrement votre niveau, ainsi que les notes suivantes.

Des éléments pris en compte lors de la notation :

- vous avez respecté ce qu'on demande (la date limite pour l'envoi de votre code et compte-rendu, le format de la soumission etc.)
- votre code contient des **commentaires** qui justifient bien le rôle de chaque classe, membre de classe, méthode, et qui expliquent les détails moins évidents d'implémentation
- le compte-rendu est un document soigné et bien organisé (introduction, rappel bref de l'objectif, contenu, discussion finale) qui justifie tous vos **choix** de design des classes (i.e. pour quoi ce membre est initialisé par défaut ainsi, pour quoi l'allocation de ce tableau se fait dans cette méthode et pas dans le constructeur etc.), mais on ne met ni les détails d'implémentation (pour cela on a les commentaires), ni des copier-coller des classes et des méthodes entières pour faire du volume
- la syntaxe de votre programme est **correcte** (i.e. le code compile)
- votre programme fonctionne correctement (i.e. le résultat correspond à ce qui est demandé), les tests demandés sont implémentés
- votre code est de bonne qualité (i.e. pas de fuites de mémoire, listes d'initialisation pour les constructeurs, encapsulation efficace, utilisation correcte des attributs vus en cours (public, private, const, static etc.) bonne organisation du code en méthodes)

1 Implementation des conteneurs pour des matrices

Le but de ce TP est de réaliser deux variantes différentes de classes pour la manipulation de matrices. Le fait d'avoir des implémentations différentes pour faire les mêmes opérations est une illustration de l'héritage vu en cours et des mécanismes de polymorphisme.

Exercice 1 — La classe abstraite de base

Regardez la classe `SafeMatrix` fournie en `SafeMatrix.h` et avant de continuer, essayez de justifier (pour vous mêmes et aussi dans votre compte-rendu plus tard) :

1. est-ce qu'on peut créer un objet de type `SafeMatrix` dans notre programme ?
2. quel est le type de données sauvegardées dans la matrice et pour quoi on le définit dans un macro en début de la classe ?

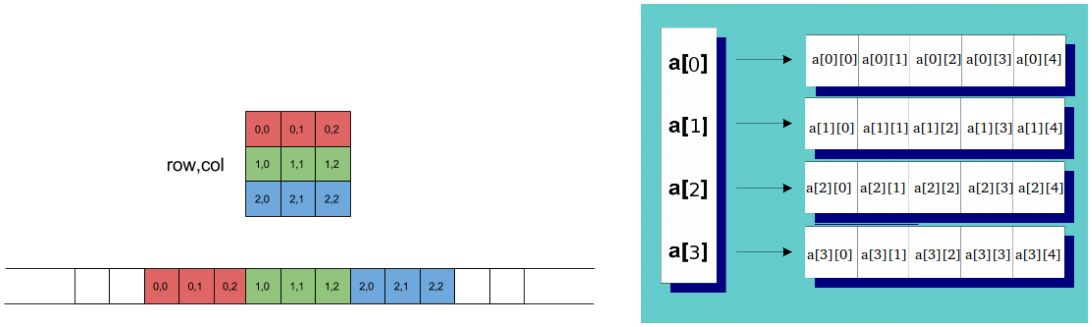

FIGURE 1 – a) matrice linéarisée ; b) matrice à double indirection

3. qu'est ce que la présence de `= 0` à la fin d'une signature de méthode impose aux classes qui héritent de cette classe de base ?
4. quelle est la différence essentielle entre les méthodes `get` et `at` qui justifie qu'une soit publique et l'autre privée ?
5. comment la surcharge de l'opérateur `<<` fonctionne pour que chaque classe dérivée affiche des choses différentes ?

Exercice 2 — Representation linéarisée d'une matrice

Par la suite, on va définir une implémentation `LinearMatrix` de la classe abstraite `SafeMatrix`, sous la forme d'une matrice linéarisée : les valeurs de la matrice sont gardées, ligne après ligne, dans un grand tableau - voir la Figure a).

1. Quels sont les membres privés nécessaires pour cette classe ?
2. Écrire un constructeur vide, un constructeur à deux paramètres, un constructeur de copie, l'opérateur $=$, le destructeur (important car on fait de l'allocation dynamique !) et la méthode `print` pour afficher le contenu de la classe
3. Écrire les deux méthodes `get` et `at`, et modifier éventuellement les constructeurs et méthodes précédentes pour que tous les accès aux données de la matrice se passent **exclusivement** à travers `get` et `at`. Ces deux dernières méthodes doivent effectuer un test pour s'assurer que les indices indiqués font bien partie de l'intérieur de la matrice (en utilisant un `assert`)
4. Écrire la méthode `void fill(typevar val)` qui remplit la matrice avec la valeur indiquée
5. Enfin, ecrire deux méthodes `LinearMatrix add ( const LinearMatrix& param ) const` et `LinearMatrix mult ( const LinearMatrix& param ) const` qui nous permettent de realiser des opérations d'addition et de multiplication
6. Tester la classe `LinearMatrix` dans une fonction libre `void test_LinearMat()`

> **Important** : normalement, pour compiler avec succès votre classe `LinearMatrix`, vous devriez implémenter toutes les méthodes virtuelles pures demandées par la classe de base, ce qui vous contraint d'écrire des dizaines de ligne de code sans pouvoir tester ce que vous faites. Pour pouvoir compiler et tester, commentez un maximum de choses dans la classe de base et implémentez **progressivement** tout ce qui est demandé.

Exercice 3 — Representation 2D d'une matrice

Implémenter en suivant exactement les mêmes étapes une classe dérivée `DynMatrix` qui utilise cette fois pour sauvegarder les valeurs de la matrice une structure a double indirection (un tableau de tableaux) - voir la Figure b).

1. d'abord un tableau ayant le même nombre d'éléments que le nombre de lignes, ou chaque case contient l'adresse d'un autres tableau
2. chacun des tableaux indiqués dans ces cases contient tous les valeurs qui se trouvent sur une ligne de la matrices (donc il a autant d'éléments qu'il y a des colonnes dans la matrice)

Exercice 4 — Illustration du polymorphisme

Une fois que les deux classes `LinearMatrix` et `DynMatrix` sont prêtes, de-commentez la première partie de la fonction `test_poly()` fournie en `main.cpp` et interprétez ce qui se passe. Pour quoi peut-on parler de polymorphisme ?

Exercice 5 — Nettoyage et réorganisation

Si vous utilisez correctement les méthodes accesseurs, les implémentations des deux classes `LinearMatrix` et `DynMatrix` ont beaucoup de méthodes identiques, ce qui indique un gros problème de design. Déplacez ces méthodes communes dans la classe de base, pour que seulement les implémentations spécifiques restent dans les classes dérivées.

Exercice 6 — Plus de polymorphisme

Apportez les modifications nécessaires pour que vous puissiez de-commenter et compiler la deuxième partie de la fonction `test_poly()`. Expliquer en détail ce qui se passe.