L3 ISFA – Programmation
Projet Individuel – Matrices

À rendre le 18 janvier 2015

Résumé

L'objectif de ce projet est de mener à bien un projet comportant des classes, en utilisant les notions d'héritage, de pointeurs et d'allocation dynamique. Les exercices réutilisent tous des notions vues au cours des TP.

Notez bien que la division en exercices est artificielle : le projet peut se lire comme un seul long exercice.

Les exercices optionnels sont, comme leur nom l'indique, facultatifs. Ils vous apporteront néanmoins des points bonus si vous les avez réussis. Cependant, préférez rendre les exercices non-optionnels parfaitement réussis plutôt qu'un programme couvrant tout le projet mais avec des bugs.

Vous pouvez nous adresser vos questions (de préférence par mail) si vous avez des problèmes.

Consignes de rendu

Vous devrez rendre le projet avant le lundi 18 janvier 2015, 23h59 (GMT+1, heure d'hiver), à Yahia Salhi (yahia.salhi@gmail.com) et Antoine Dailly (antoine.dailly@univ-lyon1.fr). Aucun projet rendu en retard ne sera corrigé. Telle est la dure loi de la vie.

Les fichiers suivants devront être inclus dans le rendu :

— `main.cpp`
— `Matrice.h`
— `Matrice.cpp`
— `MatriceCarree.h`
— `MatriceCarree.cpp`

Le code devra être soigneusement commenté.

L'objet de votre mail de rendu devra être : [Rendu Projet Matrice] NOM Prénom.

□

Modalités d'évaluation

Votre projet sera évalué sur les points suivants :

— Bon fonctionnement dans des conditions normales ;
— Robustesse (bon fonctionnement dans les cas extrêmes) ;
— Qualité et lisibilité du code source ;
— Bonne compréhension et utilisation des concepts vus pendant le semestre.

□

Exercice 1 : Création du projet

Le but de ce projet est de disposer d'une classe `Matrice` implémentée par vos soins, et possédant diverses méthodes permettant de la manipuler.

Commencez par créer un fichier `main.cpp`, qui vous permettra de tester votre implémentation.

Créez ensuite un fichier `Matrice.h` et son fichier associé `Matrice.cpp`.

Exercice 2 : La classe `Matrice`

Notre classe `Matrice` désigne des matrices $m \times n$. Elle doit être implémentée de façon dynamique (utilisation de pointeurs). Elle possède donc trois attributs :

— `_m`, un entier qui désigne le nombre de lignes de la matrice ;

— `_n`, un entier qui désigne le nombre de colonnes de la matrice ;
— `contenu`, le contenu de la matrice en question, qui doit pouvoir contenir des `double`.

Les attributs `_m` et `_n` doivent pouvoir être lus et modifiés par des accesseurs. Pour `contenu`, on souhaite pouvoir lire une valeur donnée dans la matrice, et modifier une valeur donnée dans la matrice.

Créez le squelette de la classe `Matrice`, ainsi que ses accesseurs dont voici les signatures :
— `int getM() const;`
— `int getN() const;`
— `void setM(int const& m);`
— `void setN(int const& n);`
— `double getValue(int const& i, int const& j) const;`
— `void setValue(int const& i, int const& j, double const& x);`

Exercice 3 : Constructeurs et destructeur

Créez les constructeurs de la classe `Matrice`. On souhaite avoir un constructeur vide, un constructeur qui crée simplement la matrice sans initialiser ses valeurs, un constructeur qui initialise chaque case de la matrice selon une valeur passée en paramètre, ainsi qu'un constructeur par copie. Ces constructeurs devront utiliser `malloc` afin de créer l'attribut `contenu`.

Créez également le destructeur de la classe `Matrice`. Il devra utiliser `free` afin de libérer la mémoire réservée par l'attribut `contenu`.

Les signatures de ces fonctions sont données ci-dessous :
— `Matrice();`
— `Matrice(int const& m, int const& n);`
— `Matrice(int const& m, int const& n, double const& x);`
— `Matrice(Matrice const&);`
— `virtual ~Matrice();`

Exercice 4 : IO

Créez les méthodes `void affecter();` et `void afficher() const;`. La première permet à l'utilisateur de renseigner lui-même les valeurs de la matrice, et la deuxième affiche la matrice.

Exercice 5 : Sous-Matrice

Créez la méthode suivante :
— `Matrice sousMatrice(int const& i, int const& j, int const& k, int const& l) const;`

Cette méthode renverra la sous-matrice comprise entre les lignes `i` et `j` et entre les colonnes `k` et `l`.

Exercice 6 : Transposition

Créez la méthode `transpose`, qui crée la transposée de votre matrice. Sa signature sera `Matrice transpose() const;`.

Exercice 7 : Produit de Kronecker

Créez la méthode `kronecker`, qui crée le produit de Kronecker de votre matrice avec une autre matrice passée en paramètre.

Pour rappel, le produit de Kronecker d'une matrice A de taille $m \times n$ et d'une matrice B de taille $p \times q$ donne la matrice de taille $mp \times nq$ suivante :

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}$$

où $a_{11}B$ désigne la multiplication de B par le scalaire a_{11}.
Cette fonction aura la signature suivante : `Matrice kronecker(Matrice const& B) const;`.

Exercice 8 : Surcharge d'opérateurs

Surchargez les opérateurs suivants :
— Égalité de deux matrices ;
— Inégalité de deux matrices ;
— Addition d'un scalaire à une matrice ;
— Addition de deux matrices ;
— Soustraction d'un scalaire à une matrice ;
— Soustraction de deux matrices ;
— Multiplication d'une matrice par un scalaire ;
— Multiplication de deux matrices ;

Ces fonctions auront pour signatures :
— `bool operator==(Matrice const& A, Matrice const& B);`
— `bool operator!=(Matrice const& A, Matrice const& B);`
— `Matrice operator+(Matrice const& A, double const& x);`
— `Matrice operator+(Matrice const& A, Matrice const& B);`
— `Matrice operator-(Matrice const& A, double const& x);`
— `Matrice operator-(Matrice const& A, Matrice const& B);`
— `Matrice operator*(Matrice const& A, double const& x);`
— `Matrice operator*(Matrice const& A, Matrice const& B);`

Vous êtes autorisés (et même encouragés !) à réutiliser des méthodes et fonctions définies précédemment, et à créer des méthodes accessoires qui vous faciliteront la tâche (comme par exemple une méthode `bool memeTaille(Matrice const& A) const;` qui permettra de savoir si la matrice `A` a la même dimension que notre matrice).

Exercice 9 : Les opérateurs +=,...

Créez les méthodes suivantes :
— `Matrice& operator+=(double const& x);`
— `Matrice& operator+=(Matrice const& A);`
— `Matrice& operator-=(double const& x);`
— `Matrice& operator-=(Matrice const& A);`
— `Matrice& operator*=(double const& x);`

Une fois ceci effectué, modifiez le code des opérateurs `+`, `-` et `*` développés dans l'Exercice 8 (conservez le code précédent en commentaire) pour utiliser ces nouveaux opérateurs. Attention, tous les opérateurs de l'Exercice 8 ne seront pas modifiables !

Exercice 10 : Matrices carrées

Créez maintenant un fichier `MatriceCarree.h` et son fichier source associé `MatriceCarree.cpp`. Définissez une classe `MatriceCarree` qui hérite de la classe `Matrice`. Redéfinissez ses constructeurs, en faisant notamment en sorte que le constructeur `MatriceCarree(int const& n);` crée la matrice identité I_n.

Exercice 11 : Inversibilité, inversion

Créez les méthodes suivantes (associées à la classe `MatriceCarree`) :
— `bool inversible() const;`
— `MatriceCarree inverse() const;`

La méthode `inversible()` renverra `true` si et seulement si la matrice est inversible, et la méthode `inverse()` renverra l'inverse de la matrice.

Pour cette dernière méthode, vous pouvez adapter le programme d'inversion par Gauss-Jordan réalisé en TP. N'hésitez pas à créer des méthodes accessoires pour les classes `Matrice` ou `MatriceCarree` qui pourront vous faciliter la tâche.

Exercice (optionnel) 12 : Diverses propriétés

Toujours dans la classe `MatriceCarree`, créez les méthodes suivantes :

— `bool symetrique() const;`

— `bool positive() const;`

La méthode `symetrique()` renverra `true` si et seulement si la matrice est symétrique, et la méthode `positive()` renverra `true` si et seulement si la matrice est définie positive.

Exercice (optionnel) 13 : Factorisation de Cholesky

Toujours dans la classe `MatriceCarree`, créez la méthode suivante :

— `MatriceCarree cholesky() const;`

Cette méthode renverra, si la matrice est symétrique et définie positive, sa factorisation de Cholesky, c'est à dire la matrice triangulaire inférieure L telle que la matrice est égale à LL^T. Des algorithmes expliquant le principe de la factorisation sont disponibles sur diverses sources (comme Wikipedia).

Exercice (optionnel) 14 : Déterminant

Toujours dans la classe `MatriceCarree`, créez la méthode suivante :

— `double determinant() const;`

Cette méthode renverra le déterminant de la matrice.

Exercice 15 : Menu

Dans votre `main.cpp`, créez un menu permettant à l'utilisateur de créer une ou plusieurs matrices, et d'effectuer les diverses opérations implémentées lors de ce projet à l'aide d'une boucle `do { } while ()` et d'un `switch`. Vous stockerez les matrices dans une collection hétérogène (`Matrice **tab`) pouvant contenir indifféremment objets de types `Matrice` et `MatriceCarree`, et utiliserez le mécanisme de résolution dynamique des liens (mot-clef `virtual`) afin d'appeler les méthodes de la bonne classe.

Vous serez sûrement amenés à définir de nouvelles méthodes pour vos classes. Ces nouvelles méthodes sont laissées à votre appréciation, et vous êtes encouragés à décrire en commentaire le raisonnement que vous avez suivi pour aboutir à vos choix d'implémentation.