EFREI

L2 TP2 C++

Surcharge d'opérateurs, Variable et méthode statique

Exercice 1 : Surcharge d'opérateurs dans une classe Point

La classe Point est décrite ci-après :

```
class Point
{
public:
   Point(int a = 0, int b = 0);
   Point operator+ (const Point & p) const;
   Point & operator+= (const Point & p);
   friend ostream & operator<< (ostream & os, const Point & p);
private:
   int x;
   int y;
};
```

1) Est-il nécessaire de faire figurer dans la classe le constructeur de copie et l'opérateur d'affectation? Justifier la réponse et écrire leur code si nécessaire.

2) Ecrire le code calculant la distance de 2 points de deux manières différentes :

   - sous la forme d'une **fonction membre d1** de la **classe Point**.
   - sous la forme d'une **fonction amie d2**

Pour chaque question, écrire les codes des déclarations prototypes et des définitions des fonctions/opérateur (si besoin est) en précisant dans quels fichiers ils figurent. Indiquer, les fichiers d'entête à inclure.

1. rappel: étant donnés deux points P1(x1, y1) et P2(x2, y2) dans le plan cartésien, leur distance est exprimée par la formule : **√ ((x2 - x1)^2 + (y2 – y1)^2),** où le symbole **√** dénote la **racine carrée.**

Exercice 2 : Surcharge d'opérateurs dans une classe Point (suite)

*La **classe Point** de l'exercice 1 est prise en compte.*

On se demande comment le code suivant pourrait afficher : **b = (11, 15) puis b = (24, 35).**

```
Point a(10,20);
Point b(4, 15);
b = 7 + b;
cout << " b = " << b << endl;
b += a + 3;
cout << " b = " << b << endl;
```

Après avoir, si nécessaire, compléter le code (déclaration prototype et définition des surcharges d'opérateurs), expliquer les instructions **b = 7 + b;** et **b += a + 3;** de la manière la plus détaillée possible en précisant l'ordre des appels des opérateurs mis en jeu.

Exercice 3 : Surcharge d'opérateurs dans une classe Rationnel

L'objectif est d'écrire une **classe Rationnel** représentant des nombres rationnels dont le **numérateur et le dénominateur sont tous les deux positifs**.

Un nombre rationnel peut être **simplifié** de telle sorte que le numérateur et le dénominateur soient des entiers premiers entre eux. Le rationnel est sous **forme irréductible**.
Pour cela, on supposera que la **fonction pgcd** existe :

int pgcd(int a, int b) { return ((!b) ? a : pgcd(b, a%b)); }

exemple : pgcd(70,45) = 5; donc 70/45 = (70/5) / (45/5) = 14/9

Implémentation demandée de la classe Rationnel :

- deux attributs : le numérateur noté _num et le dénominateur _den
- un seul constructeur
- un opérateur ami pour l'affichage d'un rationnel
- la fonction simplifier
- un opérateur ami pour la somme de 2 rationnels tel que la somme soit un rationnel sous **forme irréductible**
  attention : cette fonction est à effet de bord, elle modifie éventuellement les 2 rationnels en les mettant sous **forme irréductible**.
- un opérateur ami pour l'opérateur += dont le code utilise l'opérateur +.

- un opérateur ami permettant de multiplier un rationnel par un double

L'implémentation de la **classe Rationnel** doit être telle que le code suivant compile, s'éxécute en affichant le résultat demandé. Aucune définition de fonction /opérateur ne doit figurer dans le fichier d'entête . Il convient de bien séparer l'interface de la classe dans **rationnel.h** de son implémentation dans **rationnel.cpp**.

```
int main(void)
{
   Rationnel r1;
   Rationnel r2(3);
   Rationnel r3(70,45);
   Rationnel r4 = 8;

   cout << r1 << endl;
   cout << r2 << endl;
   cout << r3 << endl;
   cout << r4 << endl;
   cout << endl;

   r1.simplifier();
   r2.simplifier();
   r3.simplifier();
   r4.simplifier();
   cout << endl;
   cout << r1 << endl;
   cout << r2 << endl;
   cout << r3 << endl;
   cout << r4 << endl;
   cout << endl;

   cout << " r1 * 4.8 = " << r1 * 4.8 << endl;
   cout << " r2 * 4.8 = " << r2 * 4.8 << endl;
   cout << " r3 * 4.8 = " << r3 * 4.8 << endl;
   cout << " r4 * 4.8 = " << r4 * 4.8 << endl;
   cout << endl;

   Rationnel r5(18,15);
   r3 += r5;
   cout << r5 << endl;
   cout << r3 << endl;
   cout << endl;

   Rationnel r6, r7;
   double x = r6(24,8);
   double y = r6(54,8);
   cout << "x = " << x << endl;
   cout << "y = " << y << endl;
   return 0;
}
```

EXECUTION :

```
0
3
70/45
8

0
3
14/19
8

 r1 * 4.8 = 0
 r2 * 4.8 = 14.4
 r3 * 4.8 = 7.46667
 r4 * 4.8 = 38.4

6/5
124/45

x = 3
y = 6.75
```

Exercice 4 : Surcharge d'opérateurs dans une classe Chaîne de caractères

Concevoir une **classe CString** permettant de manipuler des chaînes de caractères et pouvant être utilisée avec le programme suivant.

1) Version 1 : purement objet

Les chaînes de caractères sont de taille fixe. Il n'est donc pas nécessaire d'écrire le constructeur par recopie ou de surcharger l'opérateur d'affectation.

```
void main()
{
// Surchages des constructeurs
// chaîne par défaut "q"
        CString s1("azerty");
        CString s2( 'q' ),
        CString s3;
        CString s4(s1);

        cout << "s1 = " << s1 << endl ;
        cout << "s2 = " << s2 << endl ;
        cout << "s3 = " << s3 << endl ;
        cout << "s4 = " << s4 << endl ;

        s3 = s1;
        cout << "s1 = " << s1 << endl ;
        cout << "s3 = " << s3 << endl ;

        // concaténation de s1 et s2
        // s1 et s2 invariantes
        s3 = s1.plus( s2 ) ;
        cout << "s1 = " << s1 << endl ;
        cout << "s2 = " << s2 << endl ;
        cout << "s3 = " << s3 << endl ;
```

```
s3 = s1.plus( 'w' ) ;
cout << "s1 = " << s1 << endl ;
cout << "s3 = " << s3 << endl ;

// si s1 > s2 au sens lexicographique
if (s1.plusGrandQue(s2) )
        cout << s1 <<  " > " << s2 << endl;
else
        cout << s1 <<  " <= " << s2 << endl;

// si s1 <= s2 au sens lexicographique
if (s2.infOuEgale(s3) )
        cout << s2 <<  " <= " << s3 << endl;
else
        cout << s2 <<  " > " << s3 << endl;
}
```

2) Version 2 : surcharge d'opérateurs

Pour rendre le programme ci-dessus plus facilement utilisable, le recours à la surcharge des opérateurs est nécessaire.

De plus les chaînes de caractères sont allouées dynamiquement, par suite le constructeur de copie doit être programmé en conséquence ainsi que la surcharge de l'opérateur d'affectation.

Le programme s'écrit alors :

```
void main()
{
// Surchages des constructeurs
// chaîne par défaut "q"
        CString s1("azerty");
        CString s2( 'q' ),
        CString s3;
        CString s4(s1);

        cout << "s1 = " << s1 << endl ;
        cout << "s2 = " << s2 << endl ;
        cout << "s3 = " << s3 << endl ;
        cout << "s4 = " << s4 << endl ;

        s3 = s1;
        cout << "s1 = " << s1 << endl ;
        cout << "s3 = " << s3 << endl ;

        // concaténation de s1 et s2
        // s1 et s2 invariantes
        s3 = s1 + s2;
        cout << "s1 = " << s1 << endl ;
        cout << "s2 = " << s2 << endl ;
        cout << "s3 = " << s3 << endl ;
s3 = s1 + 'w';
cout << "s1 = " << s1 << endl ;
cout << "s3 = " << s3 << endl ;

// si s1 > s2 au sens lexicographique
if (s1 > s2)
        cout << s1 <<  " > " << s2 << endl;
else
        cout << s1 <<  " <= " << s2 << endl;

// si s1 <= s2 au sens lexicographique
if (s2 <= s3)
        cout << s2 <<  " <= " << s3 << endl;
else
        cout << s2 <<  " > " << s3 << endl;
}
```

Exercice 5 : Surcharge d'opérateurs dans une classe Vecteur

Un vecteur d'entiers est représenté par un **tableau dynamique d'entiers**, une capacité constante (le nombre maximal d'éléments qu'il peut contenir) et une taille utile (le nombre d'éléments contenus dans le vecteur).

Ecrire une classe Vecteur ayant pour fonctions membres (méthodes):

1) un constructeur à un paramètre, la capacité, dont la valeur par défaut est fixée à 100 et dont tous les éléments sont initialisés à 0
2) un constructeur de copie
3) un destructeur
4) un accesseur en lecture pour la capacité
5) une surcharge de l'opérateur d'affectation = entre deux vecteurs de même capacité
6) une surcharge de l'opérateur d'indexation [] en lecture (Rvalue)
7) une surcharge de l'opérateur d'indexation [] en écriture (Lvalue)

Il est également demandé de surcharger les opérateurs d'entrée/sortie avec des fonctions amies.

Votre programme doit être testé avec le code suivant :

```
void demo_vecteur()
{
   Vecteur a (5);
   cin >> a;
   Vecteur b (a);
   Vecteur c (5);
   cin >> c;

   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
   cout << "c = " << c << endl;

   c = a;
   a[3] = 666;
   a[-2] = 7777;  // erreur à l'execution grâce à l'assertion
   cout << "c = " << c << endl;
   cout << "a = " << a << endl;
   c[0] = 2;
   c[1] = 6;
   c[2] = 3;
   c[3] = 1;
   c[4] = 1;
   cout << "c = " << c << endl;

   Vecteur d (5);
   d = a;
   cout << "d = " << d << endl;

   const Vecteur vconst(3);
   cin >> vconst;
   vconst[2] = 666;
   cout << "vconst = " << vconst << endl;

   cout << "produit scalaire a = " << a<< " et b = " << b << endl;
   cout << produit_scalaire(a, b) << endl;
}
```

Exercice 6 : Surcharge d'opérateurs dans une classe Matrice

Une matrice d'entiers est représentée par un tableau dynamique de pointeurs de vecteurs, un nombre de lignes constant et un nombre de colonnes constant.

Ecrire une classe Matrice ayant pour fonctions membres (méthodes):

1) un constructeur à deux paramètres, les nombres de lignes et colonnes, dont les valeurs par défaut sont toutes les deux fixées à 100 et dont tous les éléments sont initialisés à 0
2) un constructeur de copie
3) un destructeur
4) des accesseurs en lecture pour les nombres de lignes et colonnes
5) une surcharge de l'opérateur d'affectation = entre deux matrices de même dimension
6) une surcharge de l'opérateur d'indexation [] en lecture (Rvalue)
7) une surcharge de l'opérateur d'indexation [] en écriture (Lvalue)
8) une surcharge de l'opérateur d'addition + entre deux matrices de même dimension
9) une surcharge de l'opérateur unaire – (opposé de)
10) une surcharge de l'opérateur de multiplication * entre deux matrices de dimensions adéquates

Il est également demandé de surcharger les opérateurs d'entrée/sortie avec des fonctions amies.

Votre programme doit être testé avec le code suivant:

```
void demo1_matrice()
{
   Matrice M1(2,3);

   cin >> M1;
   cout << M1 << endl;

   Vecteur v(3);
   v = M1[1];
   cout << v << endl;

   cout << "M1[1][2] = " << M1[1][2] << endl;

   M1[0][1] = 666;
   cout << M1 << endl;

   Matrice M2(M1);
   M1[1][1] = 0;
   cout << M2 << endl;
```

```
void demo2_matrice()
{
   Matrice M1(2,3);
   cin >> M1;
   cout << M1 << endl;

   Vecteur v(3);
   v = M1[1];
   cout << v << endl;

   M1[0] = v;
   cout << M1[0] << endl;
   cout << "M1[1][2] = " << M1[1][2] << endl;

   M1[0][1] = 666;
   cout << M1 << endl;

   Matrice M2(M1);
   M1[1][1] = 0;
```

```
    Matrice M3(4,2);
    cin >> M3;
    cout << M3;

    M3 = M1;
    M1[1][1] = 666;
    cout << "M1 = " << M1 << endl;
    cout << "M3 = " << M3 << endl;
}
```

```
    cout << M2 << endl;

    Matrice M3(4,2);
    cin >> M3;
    cout << M3 << endl;

    M3 = M1;
    M1[1][1] = 666;
    cout << "M1 = " << M1 << endl;
    cout << "M3 = " << M3 << endl;
}
```

```
void demo3_matrice()
{
    Matrice M1(3,4), M2(3,4);
    cin >> M1 >> M2;
    cout << "M1 = \t" << M1 << endl;
    cout << "M2 = \t" << M2 << endl;
    cout << "M1 + M2 = \t" << (M1 + M2) << endl;

    Matrice M3(4,3), M4(3,2);
    cin >> M3 >> M4;
    cout << "M3 = \t" << M3 << endl;
    cout << "M4 = \t" << M4 << endl;
    cout << "M3 * M4 = \t" << (M3 * M4) << endl;

    Vecteur v(3);
    cin >> v;
    cout << "produit scalaire M3[1] = " << M3[1] << " et v = " << v << endl;
    cout << produit_scalaire(M3[1], v) << endl;
}
```

Exercice 7 : Classes et membres statiques

Adapter la classe Point des exercices 1 et 2, de manière à ce qu'un point s'affiche avec ses coordonnées et le nombre d’instances de type *Point* dans le programme.