TP 2 POO C++: La surcharge des opérateurs

IMT Atlantique – TAF Robin/Ascii

Programmation des systèmes embarqués

Objectifs :
Les objectifs de ce tp sont de mettre en œuvre la surcharge des opérateurs de flux ≪ et ≫ et de quelques opérateurs arithmétiques en C ++.

Exercice 1 – Classe `Stack` avec la surcharge d'opérateurs

On veut, dans cet exercice, étendre la classe **`stackArray`** (vue en TP1) afin d'implémenter la surcharge de l'opérateur d'affectation `=` et de quelques opérateurs arithmétiques et logiques sur les piles.

Nous rappelons ci-dessous la modélisation par la classe **`stackArray`** d'une pile au moyen d'un tableau.

```
Class stackArray
{
  private :
     static const int default_size = 50; // constante de classe
     const int tailleMax;
           int head;
           int* data;
  public :
     // constructeur par défaut
     stackArray (int size = default_size) : // A compléter
};
```

1 Surcharge d'opérateurs

La **surcharge d'opérateur** permet aux opérateurs du C++ d'avoir une signbification spécifiqueb quand ils sont appliqués à des types spécifiques. Les opérateurs C++ que l'on surcharge habituellement sont :

- Affectation, affectation avec opération (`=` , `+=` , `-=` , etc) : méthode
- Incrémentation `++`, décrémentation `--` : méthode
- Opérateurs de lecture et écriture sur flux << et >> : méthode
- Opérateur arithmétiques (`+`, `-`, `/`, etc) : fonction

La **première technique** pour surcharger les opérateurs consiste à les considérer comme des méthodes membres de la classe sur laquelle ils 's'appliquent.

Le principe est le suivant :

```
A Op B se traduit par A.operatorOp(B)
t1 == t2 // équivalent à : t1.operator==(t2)
t1 += t2 // équivalent à : t1.operator+=(t2)
```

La **seconde technique** pour surcharger les opérateurs consiste à les considérer comme des fonctions externes à la classe sur laquelle ils 's'appliquent. Ces fonctions prennent les opérandes comme arguments.

Le principe est le suivant :

```
A Op B se traduit par operatorOp(A,B)
t1 + t2 // équivalent à : operator+(t1,t2)
t1 * t2 // équivalent à : operator*(t1,t2)
```

1.1 Surcharge de l'opérateur d'affectation (=)

L'opérateur d'affectation (=) est un opérateur de copie d'un objet vers un autre. L'objet affecté est déjà créé sinon c'est le constructeur de copie qui sera appelé. La forme habituelle d'opérateur d'affectation est la suivante :

```
Class T
{
  public :
     T& operator =(const T& cT)
};
```

La surcharge de cet opérateur est réalisée sous forme d'une méthode membre de la classe sur laquelle il s'applique. Il renvoie une référence sur T afin de pouvoir l'utiliser avec d'autres affectations. En effet, l'opérateur d'affectation est associatif à droite $a = b = c$ est évaluée comme $a = (b = c)$. Ainsi, la valeur renvoyée par une affectation doit être à son tour modifiable.

La définition de l'opérateur = est la suivante :

```
stackArray & stackArray::operator = (const stackArray &p)
{
  // Vérifier si c'est pas une auto-copie :
  if (this != &p)
     {
       // TODO :
       // 1. Libérer l'ancienne pile

       // 2. Réinitialiser les attributs

       // 3. Allouer une nouvelle pile

       // 4. Recopier les éléments de la pile

     }

     #ifdef DEBUG
     cout << "operator= (const stackArray&p) : " << this << endl;
     #endif
};
```

Q1 Implémentez, en utilisant la définition ci-dessus, la surcharge de l'opérateur d'affectation. Testez le fonctionnement de l'opérateur.

1.2 Surcharge de l'opérateur de flux <<

Le flot `cout` est un **flot de sortie** prédéfini de type `ostream` connecté à la sortie standard stdout. De même, le flot `cin` est un **flot d'entrée** prédéfini de type `istream` connecté à l'entrée standard `stdin`.

On souhaite surcharger l'opérateur de flux `<<` pour la classe `stackArray` afin de pouvoir afficher le nombre d'éléments et le contenu de la pile. Pour cela, il faut définir l'opérateur « << » comme méthode membre de la classe `ostream` dont `cout` est une instance. Le principe est le suivant :

```
ostream & operator << (ostream & sortie, const stackArray  &p)
{
  // Envoi sur le flot sortie des membres de stackArray en utilisant des instructions
  // de la forme : sortie << ..... << endl;
  return sortie;
}
```

NOTE 0 : Comme cet opérateur doit afficher le contenu de la pile, il doit avoir accès aux attributs `head` et `data`. Mais comme ces attributs sont privés, il faut soit déclarer cet opérateur comme **friend** de la classe `stackArray`, soit développer une méthode «get» correspondante.

Q2 Implémentez, en utilisant la définition ci-dessus, la surcharge de l'opérateur de flux de sortie << pour qu'il affiche un objet `stackArray` de la manière suivante :

Stack p1 → (size : 4, content :<10, 20, 24, 30>)

1.3 Surcharge des opérateurs logiques et d'auto-affectation

On souhaite ajouter quelques opérateurs simples pour la classe `stackArray` :

1. opérateur `==` qui teste si deux piles sont identiques ;
2. opérateur `!=` qui teste si deux piles sont différentes ;
3. opérateur `+=` qui empile une pile sur une autre.

Q3 Définissez les opérateurs nécessaires pour cette suite de la fonction `main` compile et s'exécute correctement.

```
stackArray p1, p2 // on supposera que p1 et p2 est non vides
if (p1 == p2)
  {
    cout << " oui" << endl;
  }
else
  {
    cout << " non" << endl;
  }

if (p1 != p2)
  {
    cout << " p1 différent de p2" << endl;
  }
else
  {
    cout << " p1 égal à p2" << endl;
  }
```

1.4 Réalisation des méthodes « empiler » et « dépiler » via des opérateurs

Q4 Remplacez la fonction membre « empiler » par l'opérateur `<` et la fonction membre « depiler » par l'opérateur `>`.
- `p < n` ajoute la valeur n sur la pile p
- `p > n` supprime la valeur du haut de la pile p et la place dans n.

Exercice 2 – Nombres complexes

Le but de cet exercice est d'implémenter la classe Complexe et de définir les opérateurs nécessaires pour écrire des opérations arithmétiques simples mettant en jeu des nombre complexes et réels.

1. Définissez la classe `Complexe`, en utilisant la représentation cartésienne des nombres complexes (i.e. avec deux attributs `double` représentant respectivement les parties réelle et imaginaire du nombre complexe).

2. Ajoutez à cette classe les constructeurs et destructeur nécessaires pour que le `main` suivant compile :

```
int main()
{
  Complexe defaut;
  Complexe zero (0.0, 0.0);
  Complexe un (1.0, 0.0);
  Complexe i (0.0, 1.0);
  Complexe j;
  return 0;
}
```

3. Définissez les opérateurs nécessaires (il en faudra deux, l'un interne et l'autre externe) pour que cette suite de la fonction `main` compile et s'exécute correctement :

```
cout << zero << " ==? " << defaut;
if (zero == defaut) cout << " oui" << endl;
else cout << " non" << endl;

cout << zero << " ==? " << i;
if (zero == i) cout << " oui" << endl;
else cout << " non" << endl;
```

4. Continuez avec des opérateurs arithmétiques simples (là encore, des opérateurs externes seront nécessaires) :

```
j = un + i ;
cout << un << " + " << i << " = " << j << endl;

Complexe trois(un);
trois += un ;
trois += 1.0 ;
cout << un << " + " << un << " + " << un << " = " << trois << endl;

Complexe deux(trois);
deux -= un ;
cout << trois << " - " << un << " = " << deux << endl;

trois = 1.0 + deux ;
cout << un << " + " << deux << " = " << trois << endl;
```

5. Passez ensuite aux multiplications et divisions :

```
Complexe z(i*i);
cout << i << " * " << i << " = " << z << endl;
cout << z << " / " << i << " = " << z/i << " = " << (z /=i) << endl;

Complexe k(2.0,-3.0);
z = k;
z *= 2.0;
z *= i;
cout << k << " * 2.0 * " << i << " = " << z << endl;
z = 2.0 * k * i / 1.0;
cout << " 2.0 * " << k << " * " << i << " / 1 = " << z << endl;
```

Indication : Soient $z_1 = (x_1, y_1)$ et $z_2 = (x_2, y_2)$ deux complexes ; on a
$z_1 * z_2 = (x_1 \times x_2 - y_1 \times y_2, x_1 \times y_2 + y_1 \times x_2)$
$\frac{z_1}{z_2} = (\frac{x_1 \times x_2 + y_1 \times y_2}{x_2 \times x_2 + y_2 \times y_2}, \frac{y_1 \times x_2 - x_1 \times y_2}{x_2 \times x_2 + y_2 \times y_2})$

Exercice 3 – Polynômes

Le but de cet exercice est de faire de façon propre et complète une classe permettant la manipulation de polynômes (sur le corps réels).

1. Définissez la classe `Polynome`, comme un tableau dynamique de `double` ;
   - ajoutez-y la méthode `degre()` qui donne le degré du polynôme ; définissez à cette occasion le type `Degre` utilisé pour représenté le degré d'un polynôme ;
   - Ajoutez à cette classe
     — un constructeur par défaut qui crée le polynôme nul,
     — un constructeur qui définit tout nombre réel comme un polynôme de degré zéro,
     — un constructeur permettant de construire des monômes ($a\,X^n$), en précisant le coefficient (a) et le degré (n) : `Polynome(double a, Degre n)`
     — une méthode `init` permettant de saisir un polynôme de degré quelconque avec ses coefficients.
2. Implémentez la surcharge de l'opérateur de flux de sortie << pour qu'il affiche le polynôme. Testez le fonctionnement de vos opérateurs avec la suite d'instructions de la fonction `main`.

```
int main()
{
  Polynome p (3.2, 4);
  cout << "p=" << p << endl;
  return 0;
}
```

3. Nous souhaitons surcharger l'opérateur de multiplication pour effectuer (i) la *multiplication de 2 polynômes* et (ii) la *multiplication par un réel*. Pour chacune nous disposons de 2 opérateurs : `*` et `*=`.
   (i) *multiplication de deux polynômes* :
   - pour le premier opérateur, `*`, il nous permet d'écrire $r = p*q$. La définition du prototype comme une fonction membre (interne) est : `Polynome operator*(Polynome const& q) const` ;
   - pour le second opérateur, `*=`, il nous permet d'écrire $p* = q$. La définition du prototype comme fonction membre est : `Polynome & operator*=(Polynome const& q)` ;
   Remarque : La définition de cet opérateur peut ensuite utiliser l'opérateur `*` défini ci-dessus :

   (ii) *multiplication par un réel* :
   - pour le premier opérateur, `*` : `Polynome operator*(double x) const` ;
   - pour le second opérateur, `*=` : `Polynome & operator*=(double x)` ;

4. Ajoutez les opérateurs pour l'addition et la soustraction. Pour la soustraction, faire attention à ce que les polynômes restent toujours « bien formés », c'est-à-dire que le coefficient de plus haut degré soit non nul (sauf pour le polynôme nul).

   Il pourra, à ce sujet, être utile de faire une méthode privée `wformed_poly()` qui supprime les 0 inutiles (de degré trop élevé).

5. Ajoutez les opérateurs de comparaison `==` et `!=`.

6. On s'intéresse pour finir à la division. Vous allez pour cela implémenter une méthode (privée) `divise` qui effectue la division euclidienne de deux polynômes. Cette méthode permet de calculer à la fois le quotient et le reste de la division. Elle sera donc utile pour ensuite implémenter les deux opérateurs / et %.

   L'algorithme de la division euclidienne pour les polynômes qui, étant donné deux polynômes N (numérateur) et D (dénominateur), produit les deux polynômes Q (quotient) et R (reste) tels que $N = Q * D + R$, avec le degré de R strictement inférieur à celui de D, est le suivant :

   $Q = 0$, $R = N$ et $\delta = deg(R)$ - $deg(D)$
   While ($\delta \geq 0$ and $R \neq 0$)
   a = rtop / dtop
   $Q = Q + aX^{\delta}$
   $R = R$ - $aX^{\delta} \times D$
   $\delta = deg(R)$ - $deg(D)$

   où rtop est le coefficient de plus haut degré de R et dtop celui de D.