MOOC Intro POO C++
Tutoriels semaine 3 : surcharge d'opérateurs

Les tutoriels sont des exercices qui reprennent des exemples similaires à ceux du cours et dont le corrigé est donné progressivement au fur et à mesure de la donnée de l'exercice lui-même.

Ils sont conseillés comme un premier exercice sur un sujet que l'étudiant ne pense pas encore assez maîtriser pour aborder par lui-même un exercice «classique».

Les solutions sont fournies au fur et à mesure sur les pages paires.

> Cet exercice correspond à l'exercice «*pas à pas*» page 122 de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Le but de cet exercice est de reprendre en détail un exemple illustrant les notions de constructeur et de surcharge d'opérateur.

On cherche à définir une classe `Polynome` permettant de représenter et manipuler des polynômes réels tel que, par exemple, `3X²+2x+1`.

Nous représentons un polynôme simplement par le tableau de ses coefficients : le polynome `3X²+2x+1` sera représenté par le tableau `1`, `2`, `3` (on commence pas le coefficient de degré le plus bas, `1` ici).

Définition de la classe

Commencez par ouvrir le fichier `Polynome.cc` et définissez y la classe `Polynome` contenant un tableau dynamique de nombres réels.

Ajoutez y la méthode publique `degre()`.

(solution page suivante)

Solution :

```
#include <vector>
using namespace std;

typedef size_t Degre;

class Polynome {
public:
  Degre degre() const { return p.size()-1; } // On devra garantir que p.size() n'est jamais nul !

private:
  vector<double> p;
};
```

Constructeurs

Il nous faut maintenant définir les constructeurs. Il nous faut au moins le constructeur par défaut : choisisons que ce soit le polynôme nul qui soit construit.

(solution page suivante)

Solution :

```
class Polynome {
public:
  // constructeur par défaut
  Polynome::Polynome() : p(1, 0.0) {}


  Degre degre() const { return p.size()-1; }

private:
  vector<double> p;
};
```

Revoir si nécessaire l'initialisation des `vector`.

Note : concernant le constructeur de copie, celui fourni automatiquement par le langage/le compilateur convient très bien. Il n'y a en effet rien de particulier à faire ici, autre que la copie usuelle des attributs ; ce que fait justement le constructeur de copie fourni par défaut.

On peut aussi ajouter d'autres constructeurs plus « pratiques », comme par exemple le plongement du corps des réels dans l'anneau des polynômes réels (c.-à-d. que tout nombre réel peut être vu comme un polynôme de degré 0), ce qui permet par exemple d'écrire :

```
 Polynome un_polynome(2.3);
```

On écrirait alors pour cela :

```
class Polynome {
public:
  Polynome(); // constructeur par défaut
  Polynome(double); // plongement du corps des réels
  ...
```

mais on peut faire mieux et utiliser la valeur par défaut des arguments, pour fusionner le constructeur par défaut et ce dernier constructeur.

(solution page suivante)

Solution :

```
class Polynome {
public:
  Polynome (double = 0.0);
  ...
```

Sa définition ne change pas fondamentalement du constructeur par défaut précédent :

```
Polynome::Polynome(double x) : p(1, x) {}
```

Pour finir avec les constructeurs, on peut aussi ajouter une façon de déclarer un monôme de degré quelconque. Par exemple le polynôme `3X^2` pourrait s'écrire :

```
Polynome p(3.0, 2);
```

où `3.0` est le coefficient et `2` le degré.

Ceci est possible en complétant le constructeur par défaut :

(solution page suivante)

Solution :

```
class Polynome {
public:
  Polynome(double coef = 0.0, Degre degre = 0);
  ...
```

avec pour définition

```
Polynome::Polynome(double coef, Degre deg)
 : p(deg+1, 0.0) // garantit que le degré de p est deg
{
  p[deg] = coef; // change la valeur du coef de plus haut degre
}
```

Opérateurs

Ajoutons maintenant à quelques opérateurs simples à nos `Polynomes`. Nous ne présenterons ici que la multiplication et l'affichage.

Affichage

Commençons par l'affichage, c'est-à-dire l'opérateur `<<` de la classe `ostream` (comme `cout`). Cet opérateur est donc un opérateur externe à la classe `Polynome` (puisque son opérande de gauche est `cout`, il devrait être interne à la classe `ostream`).

De plus, comme cet opérateur doit afficher le contenu du polynôme, il doit avoir accès à l'attribut `vector<double> p;`. Mais comme cet attribut est privé et que nous n'avons pas fait de méthode «get» correspondante (ce que nous ne souhaitons pas : mis à part ici, il n'y a pas besoin d'avoir accès à *tout* le tableau des valeurs), il nous faut soit déclarer cet opérateur comme `friend` de la classe `Polynome`, soit développer une méthode, publique, d'affichage que cet opérateur appelera. Choisissons cette dernière solution qui est préférable.

Essayez de le faire par vous même sans regarder la solution. (solution page suivante)

Solution :

```
#include <iostream>
...
class Polynome {
public:
  ... // comme avant

// prototype de la méthode facilitant l'affichage
void affiche_coef(ostream& out, Degre puissance, bool signe = true) const;


  ...
};

// prototype de la fonction surchargeant (externe) l'opérateur <<
ostream& operator<<(ostream&, Polynome const&);

// ---------------------------------------------------------------------
// définition de la méthode facilitant l'affichage
void Polynome::affiche_coef(ostream& out, Degre puissance, bool signe) const
{
  double const c(p[puissance]);
  if (c != 0) {
    if (signe and (c > 0.0))  out << "+";
    out << c;
    if (puissance > 1)
      out << "*X^" << puissance;
    else if (puissance == 1) out << "*X";
  } else if (degre() == 0) {
    // degré 0 : afficher quand meme le 0 si rien d'autre
    out << 0;
  }
}

// ---------------------------------------------------------------------
ostream& operator<<(ostream& sortie, const Polynome& polynome)
{
  // plus haut degré : pas de signe + devant
  Degre i(polynome.degre());
  polynome.affiche_coef(sortie, i, false);

  // degré de N à 0 : +a*X^i
  if (i > 0) {
    for (i--; i > 0; i--) polynome.affiche_coef(sortie, i);
    polynome.affiche_coef(sortie, 0);
  }

  return sortie;
}
```

On peut maintenant tester nos développements à ce stade, par exemple en ajoutant la fonction `main`

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

Multiplication

Passons maintenant la multiplication.
Il y a plusieurs cas qui nous intéressent :

1. multiplication de 2 polynômes ;
2. multiplication par un réel (sans passer par la précédente).

Pour chacune nous disposons de 2 opérateurs : `*` et `*=`. Commençons par le premier.

`*` nous permet d'écrire des choses comme `r = p * q;`, et doit donc prendre un polynôme (de plus : `q`) en argument et retourner la valeur du calcul après l'opération.
en choisissant la surcharge externe et le prototype conseillé dans la vidéo, nous avons :

```
const Polynome operator*(Polynome p, const Polynome& q)
```

La définition de cet opérateur est ensuite sans difficulté pour ceux qui connaissent la multiplication de polynômes. Comme ce n'est pas le but ici de réviser (découvrir ?) les mathématiques, nous vous le donnons directement :

```
const Polynome operator*(Polynome p, const Polynome& q) {
  const Degre dp(p.degre());
  const Degre dq(q.degre());

  // Prépare la place pour le polynome résultat (de degre p.degre()+q.degre()).
  Polynome r(0.0, dp + dq);

  // fait le calcul
  for (Degre i(0); i <= dp; ++i)
    for (Degre j(0); j <= dq; ++j)
      r.p[i+j] += p.p[i] * q.p[j];

  // retourne le resultat
  return r;
}
```

Cet opérateur ayant par contre besoin d'accéder au contenu des `Polynomes`, il est nécessaire ici de le déclarer comme `friend` :

```
class Polynome {
  //...

  // la multiplication aura besoin d'accéder à p
  friend const Polynome operator*(Polynome p, const Polynome& q);
};
```

C'est une des raisons pour lesquelle usuellement on préfère définir l'opérateur externe (`*` ici) à partir de sa version interne d'auto-affectation (`*=` ici). Mais dans le cas présent, l'algorithme utilisé pour multiplier des polynômes, même en interne, nécessiterait de toutes façons un polynôme supplémentaire (`r` du code ci-dessus). Nous avons donc choisi de commencer par la version externe de la multiplication, peut être plus naturelle.

Pour le second opérateur, `*=`, il nous permet d'écrire `p *= q;`, mais doit aussi nous permettre (norme du langage C++) d'écrire des choses comme `r = s + (p *= q);`, bien que **je vous déconseille fortement d'utiliser ce genre d'expressions**.

Donc `*=` doit aussi retourner la valeur du calcul après l'opération.

Le plus simple et le plus efficace pour ceci est encore d'utiliser les références et de choisir comme prototype :

```
Polynome& operator*=(Polynome const&);
```

La définition de cet opérateur peut ensuite utiliser l'opérateur `*` défini ci-dessus :

```
Polynome& Polynome::operator*=(Polynome const& q) {
  return (*this = *this * q);
}
```

Ce code :

- `*this * q` : fait la multiplication de l'instance courante (`*this`) par l'instance `q` en utilisant la multiplication externe précédemment définie ;
- `*this = ..` : affecte le résultat de cette multiplication à l'instance courante ;

- `return (...)` : et retourne la «valeur» (`Polynome`) en question.

Terminons maintenant par la multiplication par les réels.
Nous allons pour cela définir l'opérateur interne

```
Polynome& operator*=(double);
```

et en externe :

```
const Polynome operator*(double, Polynome const&);
const Polynome operator*(const Polynome, double);
```

qui sont utilisés respectivement pour des opérations de type `p *= x`, `q = p * x` et `q = x * p`, où `p` et `q` sont des polynômes et `x` est un double.

Leurs définitions ne présentent pas de problème majeur.

Essayez de le faire par vous même <u>sans</u> regarder la solution. (solution page suivante)

Solution :

Ces opérateurs étant liés, il est important de définir les deux derniers utilisant le premier :

```
// -----------------------------------------------------------------------
Polynome& Polynome::operator*=(double x) {
  for (Degre i(0); i <= degre(); ++i)
    p[i] *= x;
  return *this;
}

// -----------------------------------------------------------------------
const Polynome operator*(Polynome p, double x) {
  return p *= x;
}

// -----------------------------------------------------------------------
const Polynome operator*(double x, const Polynome& p) {
  return Polynome(p) *= x;
}
```

A noter qu'il n'est pas nécessaire ici que le dernier opérateur (externe) soit «`friend`» de la classe, puisqu'il ne fait appel qu'à des méthodes publiques (`operator*=` en l'occurrence).

Voilà ! Ceci conclut ce tutoriel un peu technique. Pour que nos polynômes soient pleinement intéressants, il faudrait au minimum leur ajouter l'addition... mais j'ai peur que cela fasse un peu trop `;-)`

Code source complet

(solution page suivante)

Solution :

```
#include <iostream>
#include <vector>
using namespace std;

typedef size_t Degre;

class Polynome {
public:
  // constructeurs
  Polynome(double coef = 0.0, Degre degre = 0); // monome coef * X^deg

  // méthodes publiques
  Degre degre() const { return p.size()-1; }
  void affiche_coef(ostream& out, Degre puissance,
                    bool signe = true) const;

  // opérateurs internes
  Polynome& operator*=(const Polynome& q);
  Polynome& operator*=(double);

private:
  // attributs privés
  vector<double> p;

  // la multiplication aura besoin d'accéder à p
  friend const Polynome operator*(Polynome p, const Polynome& q);
};

// opérateurs externes
const Polynome operator*(Polynome p, const Polynome& q);
const Polynome operator*(Polynome p, double x);
const Polynome operator*(double x, const Polynome& p);

// ========================================================================
// définition des méthodes

// ------------------------------------------------------------------------
Polynome::Polynome(double coef, size_t deg)
  : p(deg+1, 0.0) // garantit que le degré de p est au moins deg
{
  p[deg] = coef;
}

// ------------------------------------------------------------------------
Polynome& Polynome::operator*=(const Polynome& q) {
  /* On définit ici exceptionnellement l'opérateur interne avec
     l'opérateur externe (au lieu de faire le contraire) car on aurait
     de toutes façon besoin d'un polynôme supplémentaire pour
     effectuer ce calcul.                                              */
  return (*this = *this * q);
}

// ------------------------------------------------------------------------
Polynome& Polynome::operator*=(double x) {
  for (Degre i(0); i <= degre(); ++i)
    p[i] *= x;
  return *this;
}

// ========================================================================
// fonctions intermédiaires et opérateurs externes

// ------------------------------------------------------------------------
const Polynome operator*(Polynome p, double x) {
  return p *= x;
}

// ------------------------------------------------------------------------
const Polynome operator*(double x, const Polynome& p) {
  return Polynome(p) *= x;
}
```

```
// ----------------------------------------------------------------------
const Polynome operator*(Polynome p, const Polynome& q) {
  const Degre dp(p.degre());
  const Degre dq(q.degre());

  // Prépare la place pour le polynome résultat (de degre p.degre()+q.degre()).
  Polynome r(0.0, dp + dq);

  // fait le calcul
  for (Degre i(0); i <= dp; ++i)
    for (Degre j(0); j <= dq; ++j)
      r.p[i+j] += p.p[i] * q.p[j];

  // retourne le resultat
  return r;
}

// ----------------------------------------------------------------------
void Polynome::affiche_coef(ostream& out, Degre puissance, bool signe) const
{
  double const c(p[puissance]);
  if (c != 0.0) {
    if (signe and (c > 0.0))  out << "+";
    out << c;
    if (puissance > 1)
      out << "*X^" << puissance;
    else if (puissance == 1) out << "*X";
  } else if (degre() == 0) {
    // degré 0 : afficher quand même le 0 si rien d'autre
    out << 0;
  }
}

// ----------------------------------------------------------------------
ostream& operator<<(ostream& sortie, const Polynome& polynome)
{
  // plus haut degré : pas de signe + devant
  Degre i(polynome.degre());
  polynome.affiche_coef(sortie, i, false);

  // degré de N à 0 : +a*X^i
  if (i > 0) {
    for (--i; i > 0; --i) polynome.affiche_coef(sortie, i);
    polynome.affiche_coef(sortie, 0);
  }

  return sortie;
}

// ======================================================================
int main() {
  Polynome p(3.2, 4);
  cout << "p=" << p << endl;

  Polynome q(1.1, 2), r;

  r = p * q;
  cout << p << " * " << q << " = " << r << endl;

  r *= 2.0;
  cout << " * 2 = " << r << endl;

  return 0;
}
```