université PARIS-SACLAY

Correction TP de programmation n°7

Cours de programmation impérative

—Licence MPI L1 S2 - Info 121—

Exemple d'utilisation d'un type abstrait : les polynômes

Dans cette séance de travaux pratiques, nous utilisons une implantation du type abstrait `Polynôme`, défini par les fonctions ci-dessous :

```
1    void  PolynomeNul(Polynome &p);
2    void  modifierCoeffPoly(Polynome &p, int d, float co);
3    int   degrePoly(Polynome p);
4    float coeffPoly(Polynome p, int d);
5    bool  estNulPoly(Polynome p);
6    bool  egalPoly(Polynome p, Polynome q);
```

L'implantation est fournie dans deux fichiers `PolyAbstr.hpp` et `PolyAbstr.cpp`. Il n'est pas utile que vous compreniez le contenu de ces deux fichiers qui utilisent la programmation orientée objet.

1 Conventions de nommage

Quand on commence à écrire des programmes avec quelques dizaines de fonctions, pour éviter d'avoir en permanence à se poser des questions comme « Quel est le nom de la fonction qui ... » ou bien « Dans quel ordre dois-je passer les paramètres à la fonction ... », il est utile de suivre quelques conventions. Toutes les entreprises de développement de logiciels, tous les projets de logiciels sérieux fixent ainsi un certain nombre de règles concernant l'indentation du code, le choix des noms des fonctions et l'ordre des paramètres.

Les choix faits dans la convention suivante sont pour la plupart arbitraires, ils sont là seulement pour éviter des hésitations comme par exemple : dois-je écrire «`EstVide`» ou bien «`estVide`» où encore « `est_vide` » ? Rien ne vous interdit de vous en écarter si elles ne vous plaisent pas, mais ne pas suivre de conventions est un bon moyen de perdre bêtement du temps.

- Les noms des types commencent par une majuscule (par exemple `Polynome`).
- On utilise la convention dite `camelCase` pour les noms de fonctions (les mots accolés et commençant par une majuscule).
- Les **constructeurs d'un type abstrait** commencent par le nom du type (par exemple `PolynomeNul`) ; le premier paramètre est l'objet à construire.
- Les procédures et fonctions qui manipulent un type abstrait ont leur **nom qui se termine** par le type (ou une abréviation, ici `Poly`).

Quand une procédure fait un calcul, il y a souvent deux manières de transmettre le résultat :

1. soit on **modifie** l'un des paramètres passé en mode Donnée-Résultat ; dans ce cas le nom de la fonction sera un **verbe conjugué qui décrit l'action** que l'on fait sur le paramètre ; ce paramètre sera toujours **le premier paramètre**.
   Par exemple `ajoutePoly(p, q)` ajoute `q` au paramètre `p`.
2. soit on **retourne le résultat** dans un paramètre passé en mode Résultat ; dans ce cas le nom de la fonction sera un **nom qui décrit le résultat de l'action**. Le paramètre recevant le résultat sera alors toujours **le dernier paramètre**.
   Par exemple `sommePoly(a, b, c)` place dans `c` la somme `a+b`.

2 Mise en place

Les quatre fichiers `main.cpp`, `PolyAbstr.hpp`, `PolyAbstr.cpp` et `Makefile` sont à extraire depuis l'archive comme dans les précédents T.P.

1. Ouvrir le fichier d'en-tête `PolyAbstr.hpp` et lire la partie de ce fichier contenant la documentation des fonctions, en cherchant à bien comprendre ce que fait chaque fonction.
2. Ouvrir le fichier `main.cpp` et deviner ce qu'il va afficher à l'exécution (sans le compiler ni l'exécuter pour l'instant).
3. Comme le code est composé de plusieurs fichiers, il faut normalement taper plusieurs commandes pour le compiler. Il est plus pratique d'écrire une bonne fois pour toutes les commandes à taper et de laisser la machine lancer les commandes nécessaires à la compilation. Pour ceci, on utilise l'outil `make`. Avec cet outil, on décrit la configuration du projet dans un fichier appelé `Makefile`. Ensuite, pour compiler projet il suffit de taper dans le terminal la commande
   ```
   make
   ```
   L'outil va lancer tout seul les appels au compilateur nécessaires comme si vous les aviez tapé au clavier. Il faut donc comme d'habitude se trouver dans le dossier où se trouvent vos fichiers pour que la commande fonctionne. On peut ensuite faire le ménage en tapant
   ```
   make clean
   ```
4. Après avoir compilé en utilisant `make`, lancez le programme `main`. Vous devriez normalement voir apparaître :
   ```
   Le polynome p est : 4X^5 - 5X^2 + X - 1
   Le polynome 3*p est : 12X^5 - 15X^2 + 3X - 3
   La derivee de p est : 20X^4 - 10X + 1
   ```

Ainsi, nous pouvons créer, afficher, multiplier par une constante, et dériver un polynôme.

On remarque que pour pouvoir les utiliser, vous n'avez pas besoin de savoir comment sont implantés les polynômes. (Pour les curieux : nous utilisons ici des tables associatives qui sont usuellement elles-mêmes un type abstrait implanté avec des arbres auto-équilibrants dits rouge-noir.)

Attention : Dans la suite de ce TP, on ne modifiera que le fichier `main.cpp`. On considère en effet que le travail d'implantation du type abstrait est effectué par une autre équipe de développement. La semaine prochaine, nous ferons le contraire : on ne modifiera pas le programme principal, mais on implantera nous-même les 5 fonctions du type abstrait.

3 Calcul avec les polynômes

En s'inspirant des fonctions déjà écrites, on demande d'écrire et de tester dans le `main` les fonctions suivantes (en prenant soin de procéder à chaque étape à des tests de validation) :

1. `void polynomeCoeffEgaux(Polynome &p, int degree, float coeff)` qui construit un polynôme p de degré `degree` pour lequel tous les coefficients sont égaux à `coeff`. Ainsi, l'appel `polynomeCoeffEgaux(p, 3, 2)` a comme résultat le polynôme $2X^3 + 2X^2 + 2X + 2$.

```
1  void polynomeCoeffEgaux(Polynome &p, int degree, float coeff) {
2    PolynomeNul(p);
3    for (int i=0; i<=degree; i++)
4      modifierCoeffPoly(p, i, coeff);
5  }
```

2. `float evalPoly(Polynome p, float x)` qui calcule la valeur du polynôme p au point x. Par exemple, pour le polynôme $X^4 + 2X^2 - 5$, on s'attend à ce que la fonction `evalPoly` renvoie 19 si elle est évaluée pour $X = 2$. Vous utiliserez pour cela la fonction `puissance(float x, int k)`, qui renvoie x^k et qui est déjà fournie (voir `PolyAbstr.hpp`).
Puis (une fois la fonction `evalPoly` codée et testée) :

```
float evalPoly(Polynome p, float x) {
  float res = 0.;
  int d = degrePoly(p);
  for (int i=0; i<=d; i++)
    res += coeffPoly(p, i) * puissance(x, i);
  return res;
}
```

   (a) À l'aide de la fonction `polynomeCoeffEgaux`, construisez un polynôme de degré 10000 dont tous les coefficients sont égaux à 1.0001. Évaluez le polynôme construit à l'aide de la fonction `EvalPoly` au point $X = 1.0001$.

```
PolynomeCoeffEgaux(q, 1e4, 1.0001);
cout << "q(1.0001) = " << evalPoly(q, 1.0001) << endl;
```

   (b) On peut en fait aller beaucoup plus vite pour l'exécution de `evalPoly`. Afin d'accélérer l'évaluation d'un polynôme, nous allons implémenter une seconde fonction `evalHornerPoly(Polynome p, float x)`. Cette méthode permet de calculer la valeur d'un polynôme de degré n en ne faisant que n additions et n multiplications au lieu de $\frac{n(n+1)}{2}$. La méthode consiste à multiplier le coefficient de plus haut degré par x, puis à lui ajouter le second coefficient. On multiplie le résultat par x, auquel on ajoute le troisième coefficient, *etc.*
   Considérons l'exemple du polynôme $-2X^3 + 3X^2 - 5X + 6$, pour $X = 3$. On l'écrit sous la forme $(((-2)X + 3)X - 5)X + 6$. On effectue donc le calcul de la manière suivante :
   — Le premier coefficient (celui de plus haut degré), vaut -2.
   — Je le multiple par X et j'ajoute le deuxième coefficient soit 3, et j'obtiens -3, qui est la valeur de $(-2)X + 3$.
   — Je multiple à nouveau par X et j'ajoute le troisième coefficient -5, je passe donc à -14, qui est la valeur de $((-2)X + 3)X - 5$.
   — Je multiple une nouvelle fois par X puis j'ajoute le dernier coefficient, 6. J'obtiens le résultat final de -36 (valeur de $(((-2)X + 3)X - 5)X + 6$ soit le résultat voulu).
   Implémentez l'algorithme de Horner.

```
float evalHornerPoly(Polynome p, float x) {
  float res = 0;
  int d = degrePoly(p);
  for (int i=d; i>=0; i--)
    res = res*x + coeffPoly(p, i);
  return res;
}
```

(c) La méthode de Horner possède un autre avantage. Construisez le polynôme $X^{10} - 99X^9$, et évaluez-le pour $X = 100$ avec la méthode normale et la méthode de Horner. Que constatez-vous ?

```
PolynomeNul(q);
modifierCoeffPoly(q, 10, 1);
modifierCoeffPoly(q, 9, -99);
cout << "EvalPoly       q(100) = " << evalPoly(q, 100) << endl;
cout << "EvalHornerPoly q(100) = " << evalHornerPoly(q, 100) << endl;
```

Sans la méthode de Horner, la soustraction de deux grands nombres proches aboutit à un résultat imprécis.

3. `void ajoutePoly(Polynome &p, Polynome q)` qui ajoute `q` au polynôme `p`.

```
1  void ajoutePoly(Polynome &p, Polynome q) {
2    int d, i;
3    d = degrePoly(q);
4    for (i=0; i<=d; i++) {
5      modifierCoeffPoly(p, i, coeffPoly(p, i)+coeffPoly(q, i));
6    }
7  }
```

4. `void produitPolyXn(Polynome p, int n, Polynome &res)` qui place dans `res` le produit du polynôme p par X^n. Par exemple, si P est défini par $P = 4X^5 - 5X^2 + X - 1$, alors $P \times X^2$ est le polynôme $P = 4X^7 - 5X^4 + X^3 - X^2$

```
1  void produitPolyXn(Polynome p, int n, Polynome &res) {
2    int d, i;
3    d = degrePoly(p);
4    PolynomeNul(res);
5    for (i=0; i<=d; i++) {
6      modifierCoeffPoly(res, i+n, coeffPoly(p, i));
7    }
8  }
```

5. `void produitPoly(Polynome p, Polynome q, Polynome &res)` qui place dans `res` le produit de deux polynômes. Par exemple si P est toujours défini par

$$P = 4X^5 - 5X^2 + X - 1\,,$$

on calcule $P \cdot (X^3 + 2X - 1)$ par

$$\begin{aligned} P \cdot (X^3 + 2X - 1) &= 1 \cdot (P \cdot X^3) + 2 \cdot (P \cdot X^1) + (-1) \cdot P \cdot (X^0) \\ &= (4X^8 - 5X^5 + X^4 - X^3) + 2(4X^6 - 5X^3 + X^2 - X) + (-4X^5 + 5X^2 - X + 1) \\ &= 4X^8 + 8X^6 - 9X^5 + X^4 - 11X^3 + 7X^2 - 3X + 1 \end{aligned}$$

```
void produitPoly(Polynome p, Polynome q, Polynome &res) {
  int d, i;
  Polynome tmp;
  d = degrePoly(q);
  PolynomeNul(res);
  for (i=0; i<=d; i++) {
    produitPolyXn(p, i, tmp);
    multPolyConst(tmp, coeffPoly(q, i));
    ajoutePoly(res, tmp);
  }
}
```

6. `void puissancePoly(Polynome p, int n, Polynome &res)` qui place dans `res` la puissance n-ième du polynôme `p`. On peut alors afficher les puissances successives du polynôme $(X+1)$ et voir apparaître le triangle de Pascal :

$$
\begin{aligned}
&1\\
&X+1\\
&X^2+2X+1\\
&X^3+3X^2+3X+1\\
&X^4+4X^3+6X^2+4X+1\\
&X^5+5X^4+10X^3+10X^2+5X+1\\
&X^6+6X^5+15X^4+20X^3+15X^2+6X+1\\
&X^7+7X^6+21X^5+35X^4+35X^3+21X^2+7X+1
\end{aligned}
$$

```
void puissancePoly(Polynome p, int n, Polynome &res) {
  Polynome tmp;
  int i;
  PolynomeNul(res);
  modifierCoeffPoly(res, 0, 1.);
  for (i=0; i<n; i++) {
    tmp = res;
    produitPoly(tmp, p, res);
  }
}
```