Université Pierre et Marie Curie — **Master M1 "Mathématiques et Applications"**
Période 2 - 2011/2012 — **Programmation en C/C++**

TD 6 : Définition de classes, surcharge d'opérateurs.

Le but est d'implémenter une classe `Polynome` pour faire du calcul formel sur les polynômes (d'une variable réelle). Voici par exemple ce que l'on aimerait faire avec la classe polynôme :

```
#include <iostream>
#include "polynome.hpp"

using namespace std;
int main(void) {
    double coeff[] = { 2, 4, -1, 0, 2 };
    unsigned degre[] = { 0, 2, 3, 5, 6 }; // on suppose le tableau ordonne
    Polynome P(coeff, degre, 5);
    cout << "P:\t" << P << endl;
    Polynome Q(4, 2);
    cout << "-Q:\t" << -Q << endl;
    cout << "1+Q:\t" << 1+Q << endl;
    cout << "P+Q:\t" << P+Q << endl;
    cout << "Evaluation de P-Q en 3.14:\t" << (P-Q)(3.14) << endl;
    return 0;
}
```

Ce programme doit donner le résultat suivant :

```
P:      2 + 4x^2 + -1x^3 + 2x^6
-Q:     -4x^2
1+Q:    1 + 4x^2
P+Q:    2 + 8x^2 + -1x^3 + 2x^6
Evaluation de P-Q en 3.14:      1887.98
```

Classe `Monome`

On commence par créer la classe `Monome` qui code des monômes de la forme ax^d où a est le coefficient réel et d le degré. La définition de cette classe doit se faire dans le fichier `polynome.hpp`.

```
class Monome {
    public:
        Monome(double coeff, unsigned degre);
        unsigned degre() const;
        double coeff() const;
    private:
        double c;
        unsigned d;
};
```

1. *Ajouter les arguments par défaut* `coeff = 0` *et* `degre = 0` *au constructeur de la classe* `Monome`.
Définir dans la classe le constructeur et les méthodes constantes `degre` *(qui renvoie le degré du monôme) et* `coeff` *(qui renvoie le coefficient* `c`*). Ces méthodes seront donc* `inline`.

2. *Tester différents appels du constructeur pour définir des monômes. Le constructeur par défaut est-il défini ? Comment doit-on l'appeler ?*
Est-il nécessaire de définir le constructeur de clonage ? Le destructeur ? Si oui, faites-le.

Classe `Polynome`

Dans le même fichier `polynome.hpp`, on va définir la classe `Polynome`. On décide de représenter un polynôme de degré n par un tableau dynamique de $n+1$ monômes (même ceux de coefficient nul).

3. *Définir la classe* `Polynome` *qui contient 2 champs privés :* `n` *qui code le degré du polynôme et* `data` *destiné à contenir les* `n+1` *monômes.*
Ajouter la méthode publique `degre()` *similaire à celle de la classe* `Monome`.

4. *Définir le constructeur utilisé à la ligne* `10` *du programme de test qui servira aussi de constructeur par défaut. La définition se fera dans un fichier* `polynome.cpp`.

5. *Déclarer dans la classe et définir dans le fichier* `polynome.cpp` *les constructeurs suivants :*
- *le constructeur de la ligne* `8`
- *le constructeur de clonage/copie*
- *le constructeur qui prend une référence constante sur un* `Monome` *(permettra de convertir un* `Monome` *en* `Polynome`*).*

6. *Est-il nécessaire de définir le constructeur de clonage ? Le destructeur ? Si oui, faites-le.*

Surcharge des opérateurs (par des méthodes)

Commençons par surcharger l'opérateur d'affectation `=` de la classe `Polynome`. Si cet opérateur n'est pas défini pas le programmeur, il est généré par le système mais cette version peut-être erronée (c'est le cas pour la classe `Polynome` mais pas pour la classe `Monome`).

7. *Définir la méthode de nom* `operator=` *en complétant le code suivant.*

```
Polynome & Polynome::operator=(Polynome const & P) {
    if (&P == this) return *this;
    // sinon: recopie du polymoe P dans l'objet courant
    // si necessaire: detruire data puis reallouer
};
```

On va maintenant surcharger l'opérateur fonctionnel `()` qui prend un argument réel x et renvoie la valeur du polynome évalué en x (utilisé ligne `14`).

8. *Définir la méthode de nom* `operator()` *et de prototype* `double operator()(double x) const` *dans les 2 classes* `Monome` *et* `Polynome`*.*

9. *Définir la méthode* `operator-` *de prototype* `Polynome operator-() const` *qui surcharge l'opérateur unaire* `-` *utilisé ligne* `11`*.*

Surcharge des opérateurs (par des fonctions globales)

La surcharge de l'opérateur d'injection `<<` doit se faire par une fonction globale de prototype

```
std::ostream & operator<<(std::ostream & o, Polynome const & P)
```

Cette fonction doit renvoyer une référence sur le flux `o` modifié par l'affichage de `P`.

10. *Écrire le prototype de cette fonction précédé du mot-clé* `friend` *dans la classe* `Polynome` *(cela permet à cette fonction d'accéder au champ privé* `data`*).*
Définir cette fonction (globale et amie de la classe `Polynome`*) dans le fichier* `polynome.cpp`*. L'affichage doit s'inspirer de celui donné en exemple.*

De la même façon que pour l'opérateur `<<` on surcharge les opérateurs arithmétiques binaires `+`, `-` et `*` par des fonctions globales de nom `operator+`, `operator-` et `operator*`. Le prototype de la fonction qui surcharge l'opérateur `+` entre 2 `Polynome` est par exemple :

```
Polynome operator+(Polynome const & P, Polynome const & Q)
```

11. *Déclarer ses fonctions comme amies de la classe* `Polynome`*. Définir ces 3 fonctions dans le fichier* `polynome.cpp`*.*

12. *Tester toutes ces fonctions en utilisant le programme de test donné en exemple.*

13. *Ajouter les opérateurs de comparaison* `<` *et* `==`*. L'ordre est donné par le degré des polynômes.*

Extensions

L'idée maintenant est de mettre en évidence la souplesse de la POO. Par exemple, on peut décider de modifier la façon de stocker les différents `Monome` dans le champ `data`. Un choix serait d'utiliser un tableau dynamique dont la taille n'est pas `n+1` mais `k` où `k` est le nombre de monômes non nuls (par exemple 5 pour `P` et 1 pour `Q`). Un meilleur choix serait d'utiliser une liste chaînée de `Monome`.

14. *Changer le champ* `data` *de la classe* `Polynome` *et modifier les méthodes et opérateurs de la classe* `Polynome` *de façon à faire fonctionner le même programme de test.*