T.P. 6

Templates et polynomes

L'objectif de ce TP est de travailler sur différents aspects des modèles de classes et de fonctions. On se propose d'écrire une classe représentant la famille des polynômes à coefficients dans un anneau abstrait, et de coder plusieurs opérations sur ces polynômes.

6.1 Les polynômes

Soit $(\mathbf{A}, +, \times)$ un anneau commutatif.

On appelle *polynôme* à une variable à coefficients dans l'anneau $\mathbf{A}$ toute suite infinie $A = (a_0, a_1, \ldots, a_n, \ldots)$ d'éléments de $\mathbf{A}$ tous nuls à partir d'un certain rang :

$$\exists N \in \mathbb{N}, \quad \forall n \geq N \quad a_n = 0.$$

Tout polynôme $A = (a_0, a_1, \ldots, a_n, 0, \ldots)$ peut s'écrire :

$$A = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i.$$

Le *degré* d'un polynôme A non nul est le *plus grand* entier n tel que le coefficient a_n de x^n dans A est non nul. Ce coefficient a_n s'appelle alors le coefficient *dominant*.

L'ensemble des polynômes à une variable à coefficients dans $\mathbf{A}$ est noté $\mathbf{A}[X]$.

6.2 Premiers pas : les éléments neutres

Nous aurons besoin de vérifier régulièrement si certains coefficients des polynômes sont égaux à 0 ou à 1. Pour la majorité des classes utilisées, nous supposerons que la comparaison directe avec les entiers 0 et 1 est possible.

Question 6.1. Écrire dans le fichier `polynome.hpp` deux modèles de fonction (*templates*) `is_zero` et `is_one` qui testent si le paramètre `a` de type générique `T` est égal respectivement à 0 ou à 1.

```
template<typename T>
bool is_zero(T a){
        ...
}
```

```
6  template<typename T>
   bool is_one(T a){
8          ...
   }
```

Pour certains types flottants, à cause des erreurs d'arrondis, il se pourrait qu'après plusieurs opérations, un coefficient soit non pas égal à 0 comme il devrait, mais de l'ordre de 10^{-18}. Le test strict alors échouerait. Une manière de remédier à ce problème est d'écrire une version spécifique de ces templates pour les types concernés.

Question 6.2. Écrire une version spécifique de `is_zero` et `is_one` pour le type `double` .

Question 6.3. Écrire une version spécifique (template) de `is_zero` et `is_one` pour les types `std::complex<T>` .

6.3 La classe `Polynome<T>`

Un polynôme avec des coefficients dans `T` est représenté par un entier `n` représentant le degré, et un vecteur `coeff` de longueur `n+1` permettant de stocker les coefficients. L'élément `coeff[i]` est le coefficient de x^i si i est inférieur au degré. Par convention un degré strictement négatif (avec une préférence pour la valeur -1 pour continuer la correspondance entre la valeur de `n` et la taille de `coeff`) correspondra au polynôme nul[1].

6.3.1 Champs, constructeurs accesseurs et mutateurs

Question 6.4. Définir dans le fichier `polynome.hpp` le modèle de classe `Polynome<T>` et déclarer les champs privés `n` et `coeff` .

6.3.2 Constructeurs

Question 6.5. Écrire un constructeur par défaut qui construit le polynôme nul. Écrire un constructeur à deux paramètres qui prend en argument un coefficient `a` en référence constante et un entier `m` pour représenter le monôme ax^m. Écrire un dernier constructeur qui prend en argument un vecteur d'éléments de type `T` et produit le polynôme avec ces coefficients (on supposera que le dernier élément du vecteur est non nul).

Question 6.6. Définir une méthode accesseur `degre` renvoyant le degré du polynôme représenté par l'objet courant.

Question 6.7. Définir un accesseur à partir de l'opérateur `operator[]` pour accéder aux coefficients du polynôme. Si l'entier passé en argument est négatif ou strictement plus grand que le degré, on fera bien attention à ne pas provoquer une erreur d'accès mémoire.

1. En mathématique, la convention est que le degré du polynôme nul est $-\infty$.

6.3.3 Affichage

On veut écrire une fonction `operator<<` qui permet d'afficher un polynôme dans un flux de sortie. Cette fonction globale (template) doit être amie avec la classe `Polynome<T>` pour pouvoir accéder aux champs privés de cette classe[2]. Cette fonction d'affichage délègue une partie du travail à une fonction auxiliaire `affiche_monome` qui, comme son nom l'indique, affiche un monôme.

Question 6.8. Définir une fonction template `affiche_monome`, prenant trois arguments :
- un flux de sortie en référence `s`,
- un coefficient de type `T` (en référence constante) `a`,
- un entier `n`,

et qui affiche dans `s` le monôme de la même façon que sur les exemples suivants : `7*x^2`, `-5*x`, `42`. Rien n'est affiché si le coefficient `a` est nul.

Question 6.9. On souhaite modifier l'affichage de cette fonction en fonction du type `T`. Si ce type représente des complexes, on préférerait que la variable soit `z` plutôt que `x`. On pourrait écrire comme pour `is_zero` précédemment une spécialisation particulière de cette fonction pour ces types. On propose d'utiliser un autre mécanisme, introduit avec le standard C++ 11. La classe template `std::is_same<typename A, typename B>` définie dans l'entête `<type_traits>` a un champ booléen `value` qui vaut `true` (resp. `false`) si les types `A` et `B` sont les mêmes (resp. différents). Consultez la page de la référence de C++ consacrée à cette commande :

https://www.cplusplus.com/reference/type_traits/is_same/

Rajouter une condition dans la fonction `affiche_monome` avec ce mécanisme pour adapter la variable au type des coefficients.

On passe maintenant à la fonction d'affichage à proprement parler pour laquelle on montre plusieurs implémentations possibles.

Question 6.10. *Version extravertie.* Déclarer dans la classe `Polynome<T>` une relation d'amitié avec une fonction d'affichage template

```
template <typename U>
std::ostream & operator<<(std::ostream &, const Polynome<U> &)
```

Notez bien que les paramètres de templates de `Polynome` est différent dans le nom de la classe et le paramètre de ce `std::operator<<`. Définir cette fonction à l'extérieur de la classe. C'est la version la plus simple à écrire, mais qui ne respecte pas le principe d'encapsulation : une version détournée pourrait en principe accéder (en lecture et écriture !) aux champs privés de toutes les autres variantes de `Polynome<T>`.

On souhaite maintenant écrire une version plus stricte de cette fonction template amie : seule la version avec le paramètre `<T>` sera amie avec la classe `Polynome<T>`.

2. Grâce aux accesseurs écrits précédemment, on aurait pu faire une fonction d'affichage n'ayant pas besoin des champs privés. La question ici sert de prétexte pour travailler le point délicat de la relation d'amitié pour des classes templates.

Question 6.11. *Version intravertie.* Déclarer une relation d'amitié avec la fonction globale de même paramètre :

```
template <typename T>
class Polynome {
[...]
    friend std::ostream& operator<< <T>(std::ostream& o, const Polynome<T>&);
[...]
};
```

et écrire la définition à l'extérieur de la classe. Notez que pour que la compilation réussisse, il faut déclarer en amont, c'est à dire avant la définition de la classe `Polynome<T>` (on parle en anglais de *forward declaration*) la fonction globale `std::ostream& operator<<`, ce qui implique aussi de devoir déclarer l'existence du modèle de classe juste avant (donc deux déclarations en amont).

```
template <typename T> class Polynome;
```

Remarquons que si nous avions pu/voulu écrire le code de la fonction globale `inline` dans la classe, il n'y aurait pas eu besoin de :

— indiquer le paramètre `<T>` dans le nom de la fonction ;
— faire les déclarations en amont.

Pour plus de détails, consulter la réponse à cette question sur stackoverflow.

https://stackoverflow.com/questions/4660123/overloading-friend-operator-for-template-class/4661372#4661372

Question 6.12. Écrire dans un fichier `main.cpp` le code suivant et vérifier ainsi le bon fonctionnement de la classe `Polynome` et de méthodes jusqu'à présent définies en compilant et en exécutant la première partie de ce code (en mettant en commentaire la partie que l'on ne souhaite pas encore exécuter).

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

int main(){
//Première partie
    Polynome<double> q;
    std::cout << "Degre de q : " << q.degre() << endl;

    vector<int> v1{6, 3, 0, 1, 5};
    Polynome<int> p1(v1);

    vector<int> v2{1,0,1};
    Polynome<int> p2(v2);

    std::complex<double> a (2.0,1.0);
    std::complex<double> b (0.0,1.0);
```

```
    vector<std::complex<double>> vc{a,b};

    Polynome<std::complex<double>> pc(vc);

    std::complex<int> one = 1;
    std::cout << "Is one one : " <<  is_one(one) << endl;

    std::cout << is_zero(a) << endl;
    affiche_monome(std::cout, a, 3);
    std::cout << std::endl;

    std::cout << "p1 : " << p1 << std::endl;

//Deuxième partie
    // Somme, différence, produit
    Polynome<int> sum = p1+p2;
    Polynome<int> diff = p1-p2;
    Polynome<int> prod = p1*p2;
    std::cout << "Somme : " << sum << std::endl;
    std::cout << "Différence : " << diff << std::endl;
    std::cout << "Produit : " << prod << std::endl;

    // Division et reste
    Polynome<int> div = p1/p2;
    Polynome<int> reste = p1%p2;
    std::cout << "Quotient : " << div << std::endl;
    std::cout << "Reste : " << reste << std::endl;

    // Evaluation en un point
    std::cout << "p1(2) : "  <<  p1(2) << std::endl;

    return 0;
}
```

6.3.4 Opérations arithmétiques

Étant donnés deux polynômes à coefficients dans un même anneau (de même type `Polynome<T>`) , nous désirons en un premier temps calculer leur somme, leur différence et leur produit.

Question 6.13. Ajouter à la classe `Polynome` une méthode privée `extend` qui prend en argument un entier `m` et, si `m` est plus grand que le degré du polynôme, renvoie un « polynôme » dont le vecteur de coefficients est obtenu en copiant celui de l'objet courant, et complété avec des zéros pour avoir la taille `m+1` (c'est à dire avoir virtuellement degré `m`), sinon, il renvoie juste l'objet courant. Attention : le dernier coefficient de ce polynôme est potentiellement nul, et dans ce cas, n'est pas le coefficient dominant. Les propriétés supposées sur les objets pour garantir la cohérence de la classe ne sont donc

plus satisfaites, et on ne peut pas utiliser les polynômes ainsi créés de manière sûre sans précaution. C'est pour cela que cette méthode doit être privée.

Question 6.14. Ajouter à la classe `Polynome` une méthode privée `adjust` qui modifie un polynôme de telle sorte que son coefficient dominant ne soit pas nul. Le degré du polynôme ansi que son vecteur de coefficients sont ainsi ajustés.

Question 6.15. Déclarer et définir, en utilisant les méthodes `extend` et `adjust`, les opérateurs de somme et de différence, écrites comme des fonctions globales.

```
template <typename T>
Polynome<T> operator+(const Polynome<T> &, const Polynome<T> &);
template <typename T>
Polynome<T> operator-(const Polynome<T> &, const Polynome<T> &);
```

Remarque : comme ces opérateurs vont faire appel aux méthodes privées `extend` et `adjust`, il faudra les déclarer amis de la classe `Polynome`.

Soient $A = (a_0, a_1, \ldots, a_n, 0, \ldots)$ un polynôme de degré n et $B = (b_0, b_1, \ldots, b_m, 0, \ldots)$ un polynôme de degré m, à coefficients dans le même anneau $\mathbf{A}$. Le produit $C = AB = (c_0, c_1, \ldots, c_l, 0, \ldots)$ est un polynôme à coefficients dans $\mathbf{A}$ de degré $l = n + m$ tel que pour tout $k \in \{0, \ldots, l\}$,

$$c_k = \sum_{\substack{i=0,\ldots,n \\ j=0,\ldots,m \\ i+j=k}} a_i b_j$$

Question 6.16. Déclarer et définir la fonction globale opérateur de produit.

```
template <typename T>
Polynome<T> operator*(const Polynome<T> &, const Polynome<T> &);
```

Nous nous intéressons maintenant à la division euclidienne de deux polynômes. Si A et B sont deux polynômes à coefficients de type `T` et que B, qui est supposé non nul, a un coefficient dominant inversible dans `T` (si `T` est un type entier comme `int`, cela signifie qu'il est egal à 1 ou -1), alors il existe un unique couple de polynômes (Q, R) tels que $A = BQ + R$ et $\deg(R) < \deg(B)$. Q est alors appelé le *quotient* et R le *reste* de la division euclidienne de A par B. Par analogie avec les calculs pour les entiers, nous souhaitons écrire les opérateurs `operator/` et `operator %` renvoyant respectivement le quotient et le reste de la division du premier paramètre par le second.

Question 6.17. Écrire près de `is_zero` et `is_one` une fonction template `is_invertible` qui renvoie true si l'argument x est inversible, c'est à dire -1 ou 1 si x est de type entier, et juste non nul en général. On pourra utiliser `std::is_integral<decltype(x)>::value` qui renvoie vrai si et seulement si le type de `x` est entier (`int`, `long`, `char`, etc.)

https://www.cplusplus.com/reference/type_traits/is_integral/

Question 6.18. Écrire une fonction d'aide `euclid_div` qui prend deux polynômes A et B en argument comme référence constante et qui renvoie sous forme de paire les polynômes Q et R. Q et R sont calculés itérativement, en posant $R_0 = A$. Ã chaque itération, on calcule le monôme Q_i pour que $R_{i+1} = R_i - BQ_i$ soit de degré strictement inférieur à R_i en éliminant les termes dominants. On arrête les itérations à l'étape n telle que $\deg R_n < \deg B$. On a alors $R = R_n$ et $Q = Q_0 + \cdots + Q_{n-1}$. La fonction affiche un avertissement et renvoie la paire $(0, A)$ si le coefficient de B n'est pas inversible.

Question 6.19. Écrire les deux fonctions globales `operator/` et `operator %` qui renvoient respectivement la première et la deuxième composante de la paire renvoyée par `euclid_div` .

6.3.5 Évaluation en un point

Soit $A = (a_0, a_1, \ldots, a_n, 0, \ldots) \in \mathbf{A}[X]$ un polynôme de degré n et $x_0 \in \mathbf{A}$. Nous souhaitons évaluer le polynôme A en x_0, ce qui correspond à calculer

$$A(x_0) = \sum_{i=0}^{n} a_i x_0^i.$$

Une approche extrêmement naïve consiste à calculer les puissances de x_0, multiplier chaque puissance par son coefficient et faire ensuite la somme. Le nombre de produits nécessaires à effectuer ce calcul est $n + (n-1) + \cdots + 2 + 1 = O(n^2)$.

La *méthode de Horner* permet de réduire considérablement cette complexité, en se ramenant à une complexité en $O(n)$, en effectuant le calcul comme suit :

$$A(x_0) = ((\cdots((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + \cdots)x_0 + a_1)x_0 + a_0.$$

Question 6.20. Écrire un opérateur parenthèse () qui permet d'évaluer un polynôme en un point, sous forme de méthode de la classe `Polynome` .

Remarque : une possible implémentation de la méthode de Horner peut s'écrire en faisant appel à la fonction `std::accumulate` de la bibliothèque `numeric` . Il faut alors parcourir le vecteur des coefficient à l'envers. On pourra faire appel aux `reverse_iterator` , auxquels on peut accéder via les méthodes `rbegin()` et `rend()` . Pour plus de détails sur les `reverse_iterator` , consulter la documentation en ligne :

https://www.cplusplus.com/reference/iterator/reverse_iterator/

Question 6.21. Vous avez peut-être écrit la fonction d'évaluation pour un argument x qui aurait le même type que les coefficients du polynôme. Or on pourrait vouloir évaluer un polynôme à coefficients entiers sur un réel, ou un polynôme à coefficients réels sur une matrice carrée à coefficients complexes. Ce qui compte, c'est que si `x` est de type `U` et les coefficients sont de type `T` , il y ait des fonctions pour multiplier des éléments de type `T` et de type `U` pour obtenir quelquechose de type `U` , ce qui est le cas dans les deux exemples ci-dessus. Modifier si besoin la fonction d'évaluation. Utilisez la bibliothèque Eigen utilisée au TP2 pour calculer `p1(M)` où $M = \begin{pmatrix} 4 & 1+i \\ -2 & \sqrt{3} \end{pmatrix}$.

6.4 Autres développements

Lorsque un polynôme est représenté par le vecteur de ses coefficients, il se peut que l'occupation de la mémoire ne soit pas optimale. Pour représenter le polynôme $x^{1000} + 1$, par exemple, nous allons utiliser un vecteur de 1001 cases, alors que l'information sur ce polynôme n'est contenue que dans la première et la dernière case du vecteur et dans son degré.

Une autre façon d'encoder un polynôme alors est de le représenter comme une somme de monômes.

Question 6.22. Écrire une court modèle de classe `Monome` , avec les constructeurs et accesseurs nécessaires.

```
template <typename T>
class Monome {
private:
    int n; // degree
    T coeff; //coefficient

public:
    Monome() : ... {}
    Monome(const T& a, int m=0): ... // le monome a*x^m

    // comparaison
    friend bool operator< <T>(const Monome<T>& u, const Monome<T>& v);
    ...
}
```

Question 6.23. Écrire le modèle de classe `Polynome`

```
template <typename T>
class Polynome {
private:
    std::list<Monome> monomes;
    ...
public:
    ...
```

et y ajouter les opérations de somme, différence, produit, division, ainsi que l'évaluation du polynôme en un point.