Programmation avancée en C++: classe avancée

Emmanuel Franck

Bureau 225, UFR math info
mail: emmanuel.franck@inria.fr

1. Généralités

Rappels

- Les classes sont un outil très puissant et très utilisé qui permet de gérer facilement/efficacement des problèmes complexes.
- En mathématiques, les classes sont très adaptées aux structures algébriques et espaces vectoriels: Corps/anneaux algébriques, polynômes, matrices, vecteurs, séries de Fourier, objets géométriques.
- **Dangers**: il y a un risque de trop utiliser les classes. Il faut réfléchir a priori sur l'intérêt de la classe et sa construction.

Important

- Le constructeur par défaut, par copie et le destructeur sont régulièrement appelés implicitement. Si on les donne pas le compilateur les construit par défaut.
- C'est la même chose pour la surcharge de l'opérateur "=".
- Le constructeur par copie et le "=" donnés par le compilateur copient les membres.

Espace vectoriel des polynômes et complexe I

- La notion de classe est essentielle. On propose donc encore un exemple plus compliqué.
- On considère l' algèbre $(\mathbb{C}[X], +, *)$ des polynômes de degré inférieur à n

$$\mathbb{C}[X] = \{a_0 + a_1 X + ... + a_n X^n, (a_0, ..., a_n) \in \mathbb{C}^n\}$$

- On peut définir les opérateurs classiques. On peut aussi définir la dérivée et la primitive.
- L'espace des polynômes est un espace vectoriel normé muni du produit scalaire:

$$(P(X), Q(X)) = \int_a^b P(X)Q(X)dX$$

- Les polynômes sont très utilisés en calcul scientifique/simulation. Il s'agit donc d'un exemple classique et important.
- **Utilisation principale**: approximation/représentation de fonctions.

Espace vectoriel des polynômes et complexe II

- Théorème de Stone-Weierstrass:

$$\forall f \in C([a,b]), \quad \exists P(x) \in \mathbb{R}[X], \quad |\, f(x) - P(x) \,| \leq \epsilon$$

- f un objet d'un espace de dimension infinie, P un objet d'un espace de dimension finie (représentable par un ordinateur).
- Il existe plusieurs difficultés supplémentaires par rapport aux précédents cas:
  - La classe est basée sur $\mathbb{C}$. Donc sur une autre classe.
  - Il y a un nombre indéterminé et potentiellement important de coefficient. On va devoir gérer dynamiquement la mémoire. Les risques de bug sont plus importants.
  - On va surcharger des opérateurs supplémentaires.
  - On va donner un exemple de méthode privée.
- On pourrait définir $\mathbb{R}[X]$, $\mathbb{Q}[X]$ ou $\mathbb{N}[X]$ afin de réduire la mémoire dans certains cas.
- Faut t'il déclarer une classe à chaque fois ? On verra dans le cours 4 que non.

2. Classe et constructeurs/destruscteur

Définition de la classe

- On commence par définir par la classe dans le fichier ”hpp”.

```
1  #ifndef POLYNOME_1_HPP
2  #define POLYNOME_1_HPP
3  #include <cmath>
4  #include <iostream>
5  #include "complexe.hpp"
6  using namespace std;
7
8  class polynome {
9    int deg;
10   double a;
11   double b;
12   complexe * coefs;
13 public:
14 ....
15 }
```

- La classe est composée: du degré, du segment $[a, b]$ et un pointeur (pour le moment vide) vers les coefficients.
- On ajoute l’inclusion de ”complexe.hpp”.

Constructeur I

- On commence par le constructeur ”nul”.

```
1  polynome (){
2     deg = 0;
3     a = 0.0;   b= 1.0;
4     coefs = NULL;
5  }
```

- **Remarque**: on initialise les pointeurs à NULL pour éviter les problèmes.
- Puis on définit d'autres constructeurs.

```
1  polynome(int degree, double binf, double bsup){
2     deg = degree;
3     a = binf;   b = bsup;
4     coefs = new complexe[degree+1];
5     for(int i =0; i<deg+1;i++){
6         coefs[i] = complexe(0.0,0.0); }
7   }
```

- **Principe**: Une fois le degré connu on peut initialiser la mémoire de manière dynamique.

Constructeur II

- **Principe**: Une fois le degré connu on peut initialiser la mémoire de manière dynamique.
- **Remarques**:
  - ”complexe(0,0)” permet de construire un complexe avec le constructeur ”complexe (double reel , double img)”.
  - Le ”new” a été automatiquement surchargé pour la classe complexe.
  - On peut définir aussi un constructeur ou juste le degré est donné.
- Destructeur:

```
1 ~polynome (){
2     if(coefs != NULL){
3       delete [] coefs;
4     }
5   }
```

- **Remarque**: Exemple de destructeur non vide. Puisque la mémoire a été allouée, elle doit être libérée.
- Evidemment on teste d'abord si la mémoire est allouée avant de la libérer.
- **Essentiel**: toute mémoire allouée dans un constructeur doit être libérée dans

Constructeur III

- Constructeur par copie:

```
1 polynome (const polynome & x){
2     deg = x.deg;
3     coefs = NULL;
4     a = x.a;  b = x.b;
5     if(x.coefs != NULL){
6       coefs = new complexe[deg+1];
7       for(int i =0; i<deg+1;i++){ coefs[i] = x.coefs[i]; }
8     }
9   }
```

- Avant de copier la mémoire allouée, comme précédemment, on vérifie si la mémoire de l'objet qu'on copie est bien non-vide.
- Le constructeur par copie est créé par le compilateur si on ne le fait pas. Dans ce cas, il fait une copie "naive" de chaque membre. Exemple:

```
1 polynome (const polynome & x){
2     deg = x.deg;
3     coefs = x.coefs;
4     a = x.a; b = x.b;  }
```

Constructeur IV

- Exemple avec notre constructeur par copie:

```
1 polynome P(1);
2 polynome P2(P);
3 P.adresse();    P2.adresse();
```

```
user $ adresse :0x7fda8fc02738
user $ adresse :0x7fda8fc02808
```

- Exemple avec le constructeur par copie généré par le compilateur:

```
user $ adresse :0x7fda8fc02738
user $ adresse :0x7fda8fc02738
```

- **Défaut**: le constructeur de la compilation copie l'adresse du tableau. Donc quand l'objet P sera détruit ce sera aussi le cas pour P2.
- Exemple au tableau de l'accès mémoire.

Constructeur V

Important

La mémoire allouée par le constructeur d'un objet doit être libérée par le destructeur associé au même objet, pas par celui d'un autre objet. Pour éviter cela le constructeur par copie doit copier les données et non les adresses (versions naïves).

- Dans le cas présent, la version naïve engendre un crash lors d'opérations sur les polynômes.
- **Remarque**: On peut manipuler des adresses mais dans ce cas la classe, les constructeurs/destructeurs sont différents. Exemple. La mémoire est allouée en dehors de la classe. Le constructeur crée un lien vers la mémoire, le destructeur détruit ce lien .
- **Petit exercice**: compiler le "projet polynôme".

3. Méthodes et opérateurs

Méthodes I

- Une des premières choses à faire est de proposer les méthodes qui permettent de lire/modifier les membres privés de la classe.

```
1 complexe polynome:: get(int i){
2   complexe res;
3   res= coefs[i];
4   return res;
5 };
6
7 void polynome:: set(complexe x, int i){
8   coefs[i] = x;
9 };
```

- Premières méthodes de calcul pour $P_n(X) = \sum_{i=0}^{n} a_i X^i$ on a

$$P_n'(X) = \sum_{i=0}^{n-1} (i+1) a_{i+1} X^i, \quad \int P_n(X) = \sum_{i=1}^{n+1} \frac{1}{i} a_{i-1} X^i$$

- On définit aussi le conjugué d'un polynôme comme le polynôme avec les conjugués des coefficients.

Méthodes II

```
1  polynome polynome::conjuge(){
2    polynome res(deg);
3    for(int i=0;i<deg+1;i++){ res.coefs[i]=coefs[i].conjuge(); }
4    return res;
5  }
6  polynome polynome::derivative(){
7    polynome res(deg-1);
8    for(int i =1;i<deg+1;i++){  res.coefs[i-1]= i*coefs[i];  }
9    return res;
10 };
11 polynome polynome::primitive(){
12   polynome res(deg+1);
13   for(int i=0;i<deg+1;i++){    res.coefs[i+1]= (1.0/(i+1.0))*coefs[i]; }
14   res.coefs[0]=complexe(0.0,0.0);
15   return res;
16 }
17 complexe polynome::integral(){
18   complexe res(0.0,0.0);
19   polynome primitive;
20   primitive = this->primitive();
21   res= primitive(complexe(b,0))-primitive(complexe(a,0));
22   return res;
23 }
```

- Cela nécessite d'avoir défini le produit entier-complexe.

Surcharge opérateur I

- On commence par écrire la somme de deux polynômes:

$$P_n(X) + Q_m(X) = \sum_{i=0}^{n} a_i X^i + \sum_{i=0}^{m} b_i X^i$$

- La difficulté arrive lorsque $m \neq n$.
- **Possibilité**: cas $n = m$ simple on peut donc écrire, pour $n > m$, Q_m comme un polynôme Q_n (les coefficients supplémentaires sont nuls) puis sommer.
- On écrit une méthode pour passer de Q_m à Q_n. Cette méthode est privée. Elle n'est là que pour organiser les calculs.

```
1 class polynome {
2 .....
3 private:
4    polynome convert(int deg2){
5    if( deg2 < deg){ cout<<"conversion pas possible"<<endl; }
6     polynome res(deg2);
7     for(int i=0;i<deg+1;i++){   res.coefs[i] = coefs[i]; }
8     return res;
9   }
```

- On utilise le fait que tous les coefficients sont nuls à l'initialisation.

Surcharge opérateur II

- Plusieurs implémentations du ”+” possible dans la classe polynôme.
- La première et troisième empêchent de modifier l'objet de droite.

```
1 polynome operator + (polynome);
2 polynome operator + (polynome &);
3 polynome operator + (const polynome &);
```

- La première fait une copie (plus coûteuse) contrairement aux deux autres.

```
1   P = R + ( S + Q);
```

- La première et troisième version marchent. Pas la seconde

```
user $ main.cpp:68:9: error: invalid operands to binary expression
     ('polynome' and 'polynome')
user $ P = R + ( S + Q);
```

- **Comment ça marche ?** le code crée un objet temporaire dans lequel on stocke S+Q puis le second plus est appliqué entre R et l'objet temporaire.
- On ne peut pas prendre une référence d'un objet temporaire sauf si il est ”const”. D'où le fait que seul l'implémentation 1 et 3 sont bonnes.

Surcharge opérateur III

- Une fois cette conversion définie on peut construire l'addition:

```
1  polynome  polynome:: operator + (polynome  p){
2    int degt;
3    degt = max(deg,p.deg);
4    polynome res(degt), polynome temp;
5    if(deg < p.deg){ temp = this->convert(p.deg);
6      for(int i=0;i<degt+1;i++){
7        res.coefs[i] = temp.coefs[i] + p.coefs[i]; }
8    }
9    if(deg > p.deg){ temp = p.convert(deg);
10     for(int i =0;i<degt+1;i++){
11       res.coefs[i]= coefs[i] + temp.coefs[i];  }
12   }
13   if(deg == p.deg){
14     for(int i =0;i<degt+1;i++){
15       res.coefs[i]= coefs[i] + p.coefs[i];  }
16   }
17   return res;
18 }
```

- On convertit le polynôme de plus bas degré puis on somme. Idem pour la soustraction.

Surcharge opérateur IV

- Une fois cette conversion définie on peut construire la multiplication:

$$P_n(X)Q_m(X) = \left(\sum_{i=0}^{n} a_i X^i\right)\left(\sum_{i=0}^{m} b_i X^i\right) = \sum_{i=0}^{n+m}\left(\sum_{j}^{i} a_j b_{i-j}\right)X^i$$

```
1  polynome polynome:: operator * (polynome p){
2    int degt=deg+p.deg;
3    polynome res(degt);
4    polynome temp, temp2;
5    temp = this->convert(degt);  temp2 = p.convert(degt);
6    for(int i=0;i<degt+1;i++){
7      for(int j=0;j<i+1;j++){
8        res.coefs[i]=res.coefs[i]+temp.coefs[j]*temp2.coefs[i-j];
9      }
10   }
11   return res;
12 }
```

- On convertit les polynômes pour éviter les dépassements de tableaux.
- Une autre implémentation est possible: Tous les polynômes de degré p sont créés avec un tableau de taille $n >> p$, avec zéro dans les coefficients entre p et n. Coût de calcul plus faible mais de stockage plus important.

Surcharge opérateur IV

- On peut aussi re-définir des opérateurs moins usuels.
- Exemple 1: les opérateurs d'accès.

```
1  complexe polynome::operator [](int i){
2    return coefs[i];
3  }
4  complexe polynome::operator ()(complexe x){
5    complexe res(0.0,0.0);
6    complexe prod(0.0,0.0);
7    for(int i =0;i<deg+1;i++){
8      prod =complexe(1.0,0.0);
9      for(int j=0;j<i;j++){ prod = prod*x; }
10       res = res + coefs[i]*prod;
11   }
12   return res;
13 }
14  polynome P(1);
15  complexe x0(1.5,0.1);
16  complexe x(1,0),y(2,0);
17  P.set(x,0); P.set(y,1);
18  cout<<"coeff 0:"<<P[0]<<endl;
19  cout<<"eval P(x):"<<P(x0)<<endl;
```

Surcharge opérateur V

```
user $ coeff 0:1+i 0
user $ eval P(x0):4+i 0.2
```

- ”[]” permet d’accéder au coefficient ”i”. L’opérateur ”()” permet d’évaluer le polynôme au point x.
- On peut aussi re-définir des opérateurs moins classiques: Exemple le produit scalaire ”,”.

```
1  complexe polynome:: operator,(polynome p){
2    polynome pro, prim;
3    complexe res(0.0,0.0);
4
5    pro = (*this)*p.conjuge();
6    prim = pro.primitive();
7    res = prim.integral();
8    return res;
9  }
10 ....
11 cout<<"Produit scalaire: "<<(P,Q)<<endl;
```

Surcharge opérateur VI

- De façon naturelle, on peut définir la norme et donc les opérateurs de comparaison entre polynômes.

```
1  complexe polynome::norm2(){
2    double res =0.0;
3    complexe n(0,0);
4    n = (*this,*this);
5    res = n.module();
6    return res;
7  };
8  bool polynome:: operator > (polynome x){
9    bool res = false;
10   if(this->norm2() > x.norm2()) { res = true;}
11   return res;
12 };
13 bool polynome:: operator == (polynome x){
14   bool res = false;
15   if(this->norm2() == x.norm2()) { res = true;}
16   return res;
17 };
```

- On voit encore comment utiliser l'objet courant dans un opérateur et d'autres méthodes.

Surcharge opérateur VII

- La encore on peut surcharger les opérateurs d'entrée et de sortie. Il existe aussi plusieurs façons d'implémenter les opérateurs in/out.

```
1  ostream & operator << (ostream & out,const polynome & p){
2    for(int i=0;i<p.deg+1;i++){
3      out<<"("<<p.coefs[i]<<")"<<"x^"<<i;
4      if(i!= p.deg){ out<<" + ";}
5    }
6    return out;
7  }
8  istream & operator >>(istream & in, polynome & p){
9    for(int i=0;i<p.deg+1;i++){
10     cout<<" coef: "<<i<<endl;
11     in >> p.coefs[i];
12   }
13   return in;
14 }
```

- Il s'agit d'opérateurs extérieures amis de la classe.

Surcharge opérateur VII

- La encore on peut surcharger les opérateurs d'entrée et de sortie. Il existe aussi plusieurs façons d'implémenter les opérateurs in/out.

```
user $ coef: 0
user $ partie reel:
user $ 1.0
user $ partie img:
user $ 2.5
user $ coef: 1
user $ partie reel:
user $ 1.4
user $ partie img:
user $ 0.0
user $ (1+i 2.5)x^0 + (1.4+i 0)x^1
```

- Il s'agit d'opérateurs extérieures amis de la classe.

Opérateur affectation I

- Pour finir on considère l'opérateur d'affectation.

```
1  polynome & polynome::operator =(const polynome & p){
2    if(&p != this){
3      deg = p.deg;
4      a = p.a;
5      b = p.b;
6      if(coefs != NULL){ delete [] coefs;}
7        coefs = NULL;
8        if(p.coefs != NULL){
9          coefs = new complexe[deg+1];
10         for(int i =0;i<deg+1;i++){
11             coefs[i]=p.coefs[i];
12       }
13     }
14   }
15   return *this;
16 }
```

- **Remarque**: contrairement au constructeur par copie, lorsqu'on écrit avec $A = B$, A peut être déjà alloué.
- Par conséquent A peut contenir un polynôme de taille différente.

Opérateur affectation II

- **Stratégie**:
  - On libère la mémoire de l'objet de gauche si possible,
  - on alloue la mémoire pour un tableau de même taille que celui de l'objet de droite,
  - on copie.

- Comme précédemment on teste l'auto-affectation et si l'objet qu'on affecte n'est pas nul.
- Si il y avait auto-affectation $A = A$: on aurait libéré la mémoire de A puis essayé de copier les données de A. Pas possible car la mémoire a été libérée. Exemple typique de problème dû à l'allocation dynamique.
- **Remarque**: lorsqu'on écrit

$$R = R * P;$$

  Il ne s'agit pas d'auto-affectation car le produit créé un nouvel objet temporaire et le copie dans R.

Opérateur affectation III

- Si on ne déclare pas l'opérateur d'affectation le compilateur il crée un opérateur d'affectation du type

```
1  polynome & polynome::operator =(const polynome & p){
2    deg = p.deg;
3    a = p.a;
4    b = p.b;
5    coefs = p.coefs;
6    return *this;
7  }
```

```
1  polynome P(1);
2  complexe x(1,0),y(2,0);
3  P.set(x,0);      P.set(y,1);
4  cout<<"Polynome P: "<<P<<endl;
5  polynome Q;
6  Q=P.derivative();
7  cout<<"Polynome Q: "<<Q<<endl;
8  polynome R;
9   R = Q+P;
10 cout<<"Polynome R=P+Q: "<<R<<endl;
```

Opérateur affectation IV

- Si on utilise notre opérateur d'affectation

```
user $ Polynome P: (1+i 0)x^0 + (2+i 0)x^1
user $ Polynome Q: (2+i 0)x^0
user $ Polynome R=P+Q: (3+i 0)x^0 + (2+i 0)x^1
```

- Si on utilise celui de compilateur

```
user $ Polynome P: (1+i 0)x^0 + (2+i 0)x^1
user $ destructeur
user $ Polynome Q: (2+i 0)x^0
user $ destructeur
user $ fin addition
user $ destructeur
user $ run(91739,0x7fffa86f7340) malloc: *** error for object
     0x7f9e24d02740: pointer being freed was not allocated
```

- **Explication**: dans l'addition, on crée un objet qui contient la somme. A la fin de l'affectation l'objet qui contient la somme est détruit. Dans l'affectation on a fait pointer R sur la somme P+Q, on pointe donc sur un objet détruit.

Remarque générale

Classes et Allocation mémoire

Lorsqu'une classe contient un membre avec une allocation de mémoire dynamique (new, malloc, etc) il est essentiel pour éviter les bugs de redéfinir le constructeur par copie, l'opérateur d'affectation et le destructeur.

- **Petit exercice**: ajouter des prints dans le constructeur nul, par copie et le destructeur puis exécuter le "main".
- Il est important de se rendre compte des appels cachés à ses constructeurs/destructeurs.

Utilisation

- On peut aussi créer des tableaux dynamiques d'objet. Le "new" est automatiquement surchargé.

```
1  polynome * Ptab;
2   Ptab = new polynome[3];
```

- C'est le constructeur nul qui est appelé pour chaque objet du tableau.

```
1   Ptab[0] = P;
2   Ptab[1] = Q;
3   Ptab[2] = R;
4   cout<<"Polynome 0: "<<Ptab[0]<<endl;
5   cout<<"Polynome 1: "<<Ptab[1]<<endl;
6   cout<<"Polynome 2: "<<Ptab[2]<<endl;
7   cout<<"Polynome 0, coef 0: "<<Ptab[0][0]<<endl;
8   cout<<"Polynome 0, coef 1: "<<Ptab[0][1]<<endl;
9   cout<<" norm de P_0 :"<<Ptab[0].norm2()<<endl;
```

- On accède à un objet du tableau comme pour les types classiques. C'est compatible avec la surcharge de l'opérateur "[]".

4. Méthodes et opérateurs version 2

Défaut d'implémentation

- La classe présente un défaut d'implémentation. Exemple:

```
1 polynome operator + (polynome);
```

- On a utilisé cette implémentation pour les opérateurs.
- A chaque utilisation, l'objet en paramètre est copié.
- Pas de problème pour les petits polynomes.
- Devient couteux pour les polynomes gros: ordre elevé, dimension 2 ou > 2. La copie est un peu coûteuse. L'allocation mémoire du constructeur par copie est assez coûteuse.
- Autre implémentation: par référence.

```
1 polynome operator + (const polynome &) const;
```

- Le "+" est un opérateur binaire. Le second const indique que l'object auquel le "+" est appliqué est aussi constant.

Nouvelle classe I

- Nouvelles signatures:

```
1  polynome conjuge( const polynome & p) const;
2  polynome operator + (const polynome &) const;
3  polynome operator - (const polynome &) const;
4  polynome operator * (const polynome &) const;
5  complexe operator , (const polynome &) const;
6  complexe norm2() const;
7  bool operator <(const polynome &) const;
8  bool operator >(const polynome &) const;
9  bool operator ==(const polynome &) const;
```

- De la même Le second const indique que l'object auquel la méthode norm2 est appliqué est constant.
- **Important**: lorqu'on commence à construire une classe avec des passages par référence et des objets constant. Il faut bien le faire partout ou c'est nécessaire.

Nouvelle classe II

- **Exemple**: Norme, on oublie le second ”const” pour l'opérateur ”,”

```
1  complexe polynome:: operator,(const polynome & p) const{
2    polynome pro, prim;
3    complexe res(0.0,0.0);
4    pro = (*this)*conjuge(p);
5    res = pro.integral();
6    return res;
7  }
8  complexe polynome::norm2() const{
9    complexe n(0,0);
10   n = (*this,*this);
11   return n;
12 };
```

- Résultat à la compilation:

```
user $ polynome_2.cpp:106:5: error: no viable overloaded '='
user $   n = (*this,*this);
user $ ./complexe.hpp:37:14: note: candidate function not viable: no
     known conversion from 'const polynome' to 'const complexe' for
     1st argument
user $ complexe & operator = (const complexe &);
```

Nouvelle classe III

- Dans la première version pour l'addition (et autre) on utilisait une fonction convert.
- Elle modifiait les polynomes. Pas possible avec les "const".
- Nouvelle implémentation:

```
1  polynome convert(const polynome & p, int deg2) const{
2      if( deg2 < p.deg){ cout<<"convertion pas possible"<<endl; }
3      polynome res(deg2);
4      for(int i=0;i<p.deg+1;i++){
5              res.coefs[i] = p.coefs[i];}
6      return res;
7    }
8    polynome  polynome:: operator + (const polynome & p) const{
9    ....
10   if(deg < p.deg){ temp = convert(*this,p.deg);
11     for(int i=0;i<degt+1;i++){
12       res.coefs[i] = temp.coefs[i] + p.coefs[i];}
13   }
14   if(deg > p.deg){ temp = convert(p,deg);
15     for(int i =0;i<degt+1;i++){
16       res.coefs[i]= coefs[i] + temp.coefs[i];}
17   }
18   ....
19 }
```

5. Tests unitaires

Tests unitaires I

- Comme pour les autres classes on peut définir des tests unitaires.
- Suivant la même stratégie: un test pour les opérations arithmétiques, un autre pour les méthodes et le produit scalaire et un dernier pour les comparaisons. Exemple:

```
1  int polynome:: testu_1(){
2    polynome P(2);
3    polynome Q,R,S;
4    complexe x(1.0,1.0), y(2.0,-1.0), z(1.0,0.0);
5
6    *this = P;
7    this->set(x,0); this->set(y,1); this->set(z,2);
8    Q = this->derivative();
9    R = this->primitive();
10   S = *this+R;
11   *this = S*Q;
12
13   complexe x0(3.0,1.0), y0(8.0,-1.0);
14   if((*this)[0] == x0 && (*this)[1] == y0){ return 0;}
15   else { return 1;}
16 };
```

Tests unitaires I

- Comme pour les autres classes on peut définir des tests unitaires.
- Suivant la même stratégie: un test pour les opérations arithmétiques, un autre pour les méthodes et le produit scalaire et un dernier pour les comparaisons. Exemple:

```
1  int polynome:: testu_2(){
2    polynome P(1), Q,S;
3    complexe x(1.0,0.0), y(2.0,0.0);
4    complexe ps, ps0(4.0,0.0);
5
6    P.set(x,0); P.set(y,1);
7    *this = P;
8    Q = this->derivative();
9    ps=(*this,Q);
10   if(ps == ps0){ return 0;}
11   else { return 1;}
12
13   return 0;
14 };
```

Tests unitaires I

- Comme pour les autres classes on peut définir des tests unitaires.
- Suivant la même stratégie: un test pour les opérations arithmétiques, un autre pour les méthodes et le produit scalaire et un dernier pour les comparaisons. Exemple:

```
1  int polynome:: testu_3(){
2    polynome P(1);
3    polynome Q,S;
4    complexe x(1.0,-1.0), y(2.0,0.0);
5    bool a=false,b=false,c=false;
6
7    P.set(x,0); P.set(y,1);
8    S = P;
9    *this = P;
10   Q = this->derivative();
11   a = (*this > Q);
12   b = (Q < *this);
13   c = (S == *this);
14   if( a+b+c == 3) { return 0; }
15   else { return 1; }
16
17   return 0;
18 }
```

Conclusion

- Les classes peuvent être utilisées pour des cas plus complexes avec allocation dynamique de mémoire. Elles sont très utilisées en algèbres linéaires (matrices, vecteurs etc).
- C'est un cas plus difficile.
- Les constructeurs par défaut, par copie, le destructeur et l'opérateur d'affectation doivent être surchargés. Car ceux proposés par le compilateur ne sont souvent pas adaptés et génèrent des problèmes de gestion mémoire.

6. Polynômes 2D

Extension en dimension 2

- Extension pour voir comment gérer les tableaux de taille 2 dynamiquement:

$$P(x, y) = a + a_1 x + a_2 y + a_3 x^2 + a_4 xy + ...$$

- On peut le représenter par une matrice $(n + 1)^2$.

```
1 class polynome2d{
2   int deg;
3   double a;
4   double b;
5   complexe ** coefs;
6 public:
7 ....
8 }
```

- On définit un pointeur sur un pointeur pour gérer les tableaux de dimension 2.
- On ne détaillera pas la classe. On détaillera juste la gestion des tableaux de dimension 2.

Extension en dimension 2

- Constructeur/destructeur:

```
1  polynome2d(int degree, double binf, double bsup){
2      deg = degree;
3      a = binf;
4      b = bsup;
5      coefs = new complexe *[degree+1];
6      for(int i =0; i<deg+1;i++){
7          coefs[i] = new complexe[degree+1] ;
8          for(int j =0; j<deg+1;j++){
9                  coefs[i][j] = complexe(0.0,0.0);
10         }
11      }
12   }
13   ~polynome2d(){
14     if(coefs!=NULL){
15       for(int i=0;i<nbvar;i++){
16         if (coefs[i]!=NULL){
17           delete [] coefs[i];}  }
18     delete [] coefs;}
19   }
```

- Avec ”coefs[i][j]” on accède aux coefficients. L'opérateur (i, j) peut être redéfini pour l'accès.