Programmation avancée en C++: Héritage simple

Emmanuel Franck

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

1. Généralités

Sous groupe des complexes

- On souhaite utiliser un sous-groupe de $\mathbb{C}$ (pour la multiplication) qui sont les complexes de modules 1.
- Cela veut dire que
$$\mid a + ib \mid^2 = a^2 + b^2 = 1$$
- C'est équivalent à $\cos(\theta) + i\sin(\theta)$. Un membre de ce sous-groupe dépend juste d'un paramètre: l' angle.
- En effet on peut supposer qu'il est plus agréable d'initialiser les complexes avec cet angle ou d'assurer à la compilation qu'un complexe est nécessairement de module 1.
- On souhaite écrire une classe (avec ce nouveau paramètre θ) sans tout réécrire (opérateurs etc).
- De la même façon qu'il s'agit d'un sous-groupe de $\mathbb{C}$ héritant d'une partie des propriétés de $\mathbb{C}$, on aimerait avoir une sous-classe de la classe complexe héritant de ses propriétés.
- En POO on parle d'héritage et de classe dérivée.

Héritage principe I

Héritage

L'héritage est le mécanisme qui permet à une nouvelle classe dite ”classe dérivée” de récupérer (”hériter”) des fonctionnalités (méthodes, constructeurs, surcharge d'opérateurs, ...) d'une autre classe (classe mère) en plus de nouvelles fonctionnalités propres.

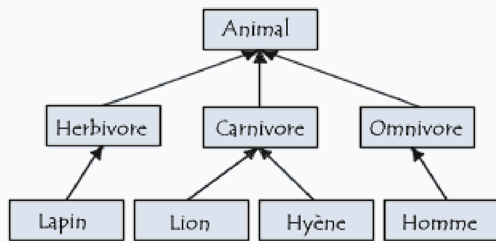
- L'héritage est essentiel en POO, on peut gagner beaucoup de temps en évitant de redéfinir ce qui vient de la classe mère.
- L'héritage est différent du fait d'utiliser une classe pour définir des membres d'une autre classe (exemple coefficient complexe dans la classe polynôme).

Important

Un objet de la classe dérivée **est** un objet de classe mère (ce n'est pas réciproque).

Héritage principe II

- Une classe mère peut avoir plusieurs classes dérivées. Il n'y a pas de limite a priori.
- Exemple:



- On peut aussi avoir plusieurs niveaux. Lapin est un objet de "herbivore" et donc "d'animal".
- L'héritage devient un outil important pour construire une spécialisation croissante.

2. Héritage exemple I: les complexes

Classe dérivée I

- On commence par voir comment on construit une classe dérivée:

```
1 class comp_unite : public complexe {
2   double theta;
3 public:
4 ...
5 }
```

- De façon assez général, lorsqu'on déclare la classe, on utilise l'expression "classe dérivée : classe mère".
- Cette forme interviendra par la suite pour les membres et constructeurs.
- Explicitation:
  - La classe "comp_unite" hérite de la classe "complexe".
  - Un objet "comp_unite" est un complexe (avec des membres "r" et "i") auquel on ajoute un membre "θ".

Important

Les membres de "complexe" étant privés, on ne peut pas y accéder à partir de la méthode de la classe dérivée. Il faut y accéder par les méthodes méthodes publiques de "complexe" (à voir plus tard).

Classe dérivée II: constructeurs/destructeurs

- On commence par le constructeur par défaut et le destructeur

```
1  class comp_unite : public complexe {
2    double theta;
3  public:
4    comp_unite (){
5      theta =0;  }
6    ~comp_unite (){ }
7  };
8  int main (){
9    comp_unite c;
10   return 0;
11 };
```

- Ici le constructeur par défaut ne traite que les nouveaux membres.

```
user $  >>>>>>> Heritage >>>>>< 
user $ constructeur complexe par défaut
user $ destructeur complexe
```

- Le constructeur et le destructeur de comp_unite appellent automatiquement ceux de ”complexe”.

Classe dérivée III: constructeurs

- Maintenant on regarde comment construire des constructeurs non-standard.

```
class comp_unite : public complexe {
  double theta;
public:
  comp_unite (){  theta =0; }
  comp_unite (double tinit): complexe(cos(tinit),sin(tinit)){
        theta = tinit;  }
....
};
```

- Pour appeler un constructeur de la classe mère, on écrit: classe_mere (....) : classe_dérivée.
- Initialiser directement *r* et *i* dans le constructeur de comp_unite car ils sont privés.

```
 comp_unite (double tinit){
    theta = tinit;
    complexe(cos(tinit),sin(tinit));
 }
```

- Implémentation fausse. Le constructeur complexe n'initialise pas l'objet courant.

Classe dérivée IV: constructeur par copie

- Maintenant on considère le constructeur par copie.
- **Question**: que se passe t'il lorsque le constructeur par copie de "complexe" où/et de "comp_unite" ne sont pas définis.
- **Cas 1**: pas de constructeur par copie pour la classe dérivée:
  - L'objet de la classe mère est considéré comme un membre de la classe dérivée donc recopié comme tous les membres.
  - S'il y a un constructeur par copie dans la classe mère il est appelé sinon on appelle celui par défaut du compilateur.
- **Cas 2**: On souhaite définir un constructeur par copie pour la classe dérivée:
- Pas d'appel automatique du constructeur par copie de la classe mère.

```
1  comp_unite (const comp_unite & x): {theta = x.theta; }
2  comp_unite cu(0.5);
3  comp_unite cu2(cu);
```

```
user $ 0.877583+i 0.479426
user $  constructeur par complexe par défaut
user $ 0+i 0
user $  constructeur par complexe par défaut
```

Classe dérivée V: constructeur par copie

- Il faut donc appeler explicitement le constructeur par copie de ”complexe” et pas n’importe comment.
- Pour le constructeur par copie, on peut procéder de la même façon.

```
1  class comp_unite : public complexe {
2    double theta;
3  public:
4    comp_unite (){  theta =0; }
5    comp_unite (double tinit): complexe(cos(tinit),sin(tinit)){
6          theta = tinit;  }
7    comp_unite (const comp_unite & x): complexe(x){
8      theta = x.theta;
9    }
10 };
```

- On appelle le constructeur par copie complexe directement dans le constructeur par copie.

```
user $ 0.877583+i 0.479426
user $ constructeur par copie complexe par défaut
user $ 0.877583+i 0.479426
```

Classe dérivée IV: utilisation

- Maintenant on écrit un exemple d'utilisation des classes dérivées

```
1  comp_unite cu(0.5);  comp_unite cu2(cu);
2  complexe c3;
3  c3 = cu +cu2;
4  cout<<"valeur du complexe: "<<c3<<endl;
```

```
user $ >>>>>>> Heritage >>>>><
user $ le + complexe
user $ le = complexe
user $ valeur du complexe: 1.75517+i 0.958851
```

- On a pu sommer des "comp_unite" et les stocker dans un complexe. Or on n'a pas défini l'addition ou le "=".
- Possible car tous les objets "comp_unite" sont des complexes.
- Les opérateurs "+" et "=" utilisés sont ceux de la classe complexe.

Utilisation classe mère par la classe dérivée

- Toutes les méthodes/opérateurs publiques de la classe mère sont utilisables par un objet de la classe dérivée.

```
cout << " >>>>>>> Heritage >>>>>< "<<endl;
 comp_unite cu(0.5);
 complexe c3;
 c3 = cu +cu2;
 cout<<"valeur du complexe: "<<c3<<endl;
 cu.modify_reel(4.0);
 double r=0;
 r = cu.module();
 bool res;  res = (cu==c3);
 cout<<" cu :"<<cu<<" res:"<<res<<" mod :"<<mod<<endl;
```

```
user $ le + complexe
user $ le = complexe
user $ valeur du complexe: 1.75517+i 0.958851

user $ cu :4+i 0.479426 res:0 mod :4.02863
```

Redéfinition des membres: opérateur affectation I

- Maintenant on considère l'opérateur d'affectation.
- **Remarque**:
  - Si l'objet est gauche est "complexe" (classe mère) le "=" de la classe mère est utilisée.
  - Si l'objet est gauche est "comp_unite" (classe fille) le "=" de la classe fille est utilisé.
- **Question**: que se passe t'il si l'opérateur d'affectation de "complexe" ou/et de "comp_unite" ne sont pas définis.
- **Cas 1**: pas de surcharge du "=" pour la classe dérivée:
  - L'objet de la classe mère est considéré comme un membre de la classe dérivée donc recopié comme tous les membres.
  - S'il y a un "=" dans la classe mère il est appelé sinon on appelle celui par défaut du compilateur. Les membres propres de la classe fille sont copiés.
  - Comportement similaire au constructeur par copie.
- **Cas 2**: On souhaite définir un constructeur par copie pour la classe dérivée.

Redéfinition des membres: opérateur affectation II

- Redéfinition de l'opérateur d'affectation.
- **Première solution**:

```
1 comp_unite & comp_unite::operator = (const comp_unite & x){
2     if(&x != this){
3       r = x.r;
4       i = x.i;
5       theta = x.theta;
6    }
7    return *this;
8 }
```

- Pas possible car "i" et "r" privés. **Deuxième solution**:

```
1 comp_unite & comp_unite::operator = (const comp_unite & x){
2     if(&x != this){
3       theta = x.theta;
4    }
5    return *this;
6 }
```

- le "=" de la classe mère ("complexe") N'EST PAS appelé, donc *r* et *i* ne sont pas recopiés.

Redéfinition des membres: opérateur affectation III

- Données de ”complexe” sont privées mais pas les méthodes. **3eme solution**:

```
1 comp_unite & comp_unite::operator = (const comp_unite & x){
2     if(&x != this){
3       this->modify_reel(x.reel());
4       this->modify_img(x.img());
5       theta = x.theta;  }
6    return *this;
7 }
```

- On utilise les méthodes public pour récupérer et modifier les membres hérités de la classe mère.

Piège

Puisque ”x” est de type constant (interdiction de modifier) les méthodes ”reel()/img()” doivent être déclarées comme ”const” aussi (elles ne peuvent pas modifier l’objet).

```
1 double complexe:: reel() const{ return r; }
2 double complexe:: img() const{  return i; }
```

Redéfinition des membres: méthodes I

- Dans une classe dérivée, on peut ajouter des nouvelles méthodes.

```
1 comp_unite comp_unite::eval(){
2   this->modify_reel(cos(theta));
3   this->modify_img(cos(theta));
4   return *this; }
5 comp_unite comp_unite::modify_theta(double thetanew){
6    theta = thetanew;
7    this->eval();
8    return *this; }
```

- Dans un classe dérivée, on peut redéfinir des méthodes de la classe mère.

```
1 double comp_unite::module(){
2    return 1.0;
3 }
4 comp_unite comp_unite::conjugue(){
5   comp_unite res;
6   theta = -theta;
7   res.eval();
8   return res;
9 }
```

Redéfinition des membres: méthodes II

- La méthode est définie dans les classes: mère et héritée. Que se passe t'il ?
- Exemple: ”le module” (en pratique pas besoin de redéfinit ”module”).

```
1  c3.module();   cu3.module();
2  bool res = (cu==c3);
3  bool res2 = (cu==cu3);
```

```
user $ module classe mère
user $ module classe héritée
user $ module classe mère
user $ module classe mère
user $ module classe mère
user $ module classe mère
```

- **Appel direct**: les objets ”complexe”/”comp_unite” utilise leurs ”module”.
- Par contre lorsque j'utilise ”==” entre deux ”comp_unite” le module ”complexe est appelé. **Pourquoi ?**
- **Réponse**: le ”==” est défini que pour la classe ”complexe” et ne connaît donc que le module ”complexe”.

Redéfinition des membres: opérateurs

- La redéfinition marche de façon similaire à celle des méthodes.
- Exemple: la multiplication, le produit de complexe de module 1 reste de module 1.

```
comp_unite compute_unite::operator * ( comp_unite){
  comp_unite res;
  res.theta = theta+x.theta;
  res.eval();
  return res;
}
int main() {
 cu3 = cu2 * cu;
 c2 = c * cu;
}
```

```
user $ multiplication classe dérivée
user $ multiplication classe mère
```

- Entre 2 objets de la classe dérivée, le "*" utilisé est celui de la classe dérivée. Sinon c'est celui de la classe mère.

Compatibilité I

- La somme de complexes de module 1 n'est pas forcément de modules 1. On ne surcharge pas l'opérateur +.

```
1  comp_unite cu(0.5);
2  comp_unite cu2(cu);
3  complexe c,c2;
4  comp_unite cu3;
5  c = cu +cu2;
6  cu3 = cu+cu2;
```

```
user $ error: no viable overloaded '='   cu3 = cu+cu2;
```

- La première somme est bonne. Le "+" étant par surchargé "cu + cu2" est de type complexe.
  - Lorsqu'on met ce résultat complexe dans un "complexe" "c" pas de problème.
  - Lorsqu'on met ce résultat complexe dans un "comp_unite" "cu3" il y a problème.
  - Comportment logique $cu + cu2$ n'est pas un complexe de module 1.

Compatibilité II

- Le fonctionnement est pointeurs est légèrement différent.

```
1 complexe d1(1.0,2.0);
2   comp_unite d2(0.2);
3   complexe * pc1 = &d1;
4   complexe * pc2 = &d1;
5   comp_unite * pcu = &d2;
6   pc1 = pcu;
7   pcu = ( comp_unite *) pc2;
```

- 1) On peut convertir un pointeur ”comp_unite” en pointeur complexe. Comme pour les objets.
- 2) L’inverse n’est pas possible pour un objet (quel serait le sens ?) mais c’est possible pour un pointeur bien que peu recommandé.
- On se concentrer sur le cas 1.
  - On a un pointeur de type complexe qui pointe sur un objet ”comp_unite”.
  - Lorsqu’on écrit $pc1->conjugue()$. Quel ”conjugue” est appelé: celui de ”comp_unite” ou de ”complexe”.
  - Le compilateur définitivement, il connaît juste le type du pointeur. Il appelle donc les méthodes ”complexe” même s’il pointe vers un ”comp_unite” .

Contrôle des accès I

Problème d'accès aux données

- Les données d'une classe sont privé (accessible seulement aux méthodes de la classe) ou public (accessible par l'utilisateur)
- Une classe dérivée et ses méthodes ne peut pas accéder aux membres privés de la classe mère.
- Cela complexifie la construction de la classe dérivée (exemple "=" pour la classe précédente, etc).
- D'un point de vue des protections des données ce n'est pas toujours nécessaire.

- Une solution très utilisée: l'accès protégé.
- Les membres protégés sont accessibles par les classes héritées, mais pas par l'utilisateur.
- On réécrit les classes "complexe et "comp_unite" à l'aide de cet accès.

Contrôle des accès II

- Définition de la classe mère:

```
1 class complexe2 {
2 protected:
3   double r;
4   double i;
5 public:
6   complexe2 (){
7     r = 0; i = 0; }
8   ......
```

- **Mot clé**: protected.
- Définition de la classe héritée:

```
1 class comp_unite2 : public complexe2 {
2   double theta;
3 public:
4   comp_unite2 (){
5     theta =0; }
6   comp_unite2 (double tinit){
7     r = cos(tinit); i = sin(tinit);
8     theta = tinit;  }
9   ......
```

Contrôle des accès III

- Suite:

```
1  comp_unite2 comp_unite2::eval(){
2    r=cos(theta);
3    i=sin(theta);
4    return *this;
5  }
6  comp_unite2 & comp_unite2::operator = (const comp_unite2 & x){
7      if(&x != this){
8        r=x.r;
9        i=x.i;
10       theta = x.theta; }
11    return *this;
12 }
```

- Dans les opérateurs/méthodes on peut accéder aux membres de la mère. Cela simplifie l'écriture.
- La plupart du temps, on utilise le mot-clé ”protected” dans les classes mère et héritée.
- L'accès totalement privé n'est utilisé que si on veut vraiment protéger les données.

Résumé

Héritage

L'héritage permet de créer des classes (dérivée) qui ont toutes les propriétés d'une autre classe (mère) + des membres et méthodes spécifiques. Pas besoin de redéfinir ce qui est dans la classe mère. Permet de spécialiser des objets.

Accès

Les classes dérivées n'ont pas accès aux données privées d'une classe mère. On utilise les méthodes publiques ou l'accès "protected".

Re-définition

On peut spécialiser des méthodes/opérateurs dans les classes dérivées.

Important

Il faut bien savoir ce qui est possible/impossible. **Important**: un objet de la classe dérivée est membre de la classe mère, pas l'inverse.

Difficulté

Le comportement des constructeurs par défaut et par copie, destructeur et opérateur "=". Toujours les redéfinir permet d'éviter les ambiguïtés.

3. Héritage exemple II: les polynomes

Interpolation et polynômes

- **En simulation, calcul scientifique** un problème classique: l'interpolation.
- On connaît une fonction $f(x)$ dans un certain nombre de points $x_0, ..., x_n$
- On veut construire un polynôme $P(x)$ tel que :

$$\mid P(x) - f(x) \mid < \epsilon$$

- On obtient d'une approximation de f à un point donné y.
- On veut pouvoir faire des calculs (arithmétique, différentiels) sur ces polynômes.

Idée

Classe dérivée: contenant les données et fonctions supplémentaires pour construire ces polynômes particuliers. Les calculs seront faits par les opérateurs/méthodes de la classe mère.

- Compatible avec les "templates".

Interpolation et polynômes I

- **Polynôme de Lagrange**: On construit de degré p tel que

$$P(x_i) = a_0 + a_1 x_i + ...a_p x_i^p = f(x_i), \forall \in \{0, ..p\}$$

- Les coefficients sont donnés en résolvant le système linéaire. Il existe une formule explicite:

$$P(x) = \sum_{j=0}^{n} f(x_i) \left(\prod_{i=0, i \neq j} \frac{x - x_i}{x_j - x_i} \right)$$

- Exemple: Degré 2

$$P(x) = f(x_0)P_0(x) + f(x_1)P_1(x) + f(x_2)P_2(x)$$

avec

$$P_0(x) = \frac{x - x_1}{x_0 - x_1} \frac{x - x_2}{x_0 - x_2}, \quad P_1(x) = \frac{x - x_0}{x_1 - x_0} \frac{x - x_2}{x_1 - x_2}, \quad P_2(x) = \frac{x - x_0}{x_2 - x_0} \frac{x - x_1}{x_2 - x_1}$$

Interpolation et polynômes II

- **Polynômes orthogonaux**: Il minimise la norme
$$\int (f(x) - P(x))^2 w(x)$$
- on obtient
$$P(x) = \sum_{k=0}^{p} c_k P_k(x) w(x)$$
avec $P_k(x)$ des polynômes de degré $k < p$ donnés par une règle de récurrence, $w(x)$ le poids. Les coefficients c_k sont définis par
$$c_k = \int_a^b f(x) P_k(x) w(x) \approx \sum_{j=0}^{n-1} (x_{j+1} - x_j) \frac{g(x_{j+1}) + g(x_j)}{2}$$
avec $g(x) = f(x) P_k(x) w(x)$ et n un nombre de points.
- On applique une formule du trapèze à chaque intervalle $[x_{j+1} - x_j]$
- Exemple: polynôme de Legendre (poids $w(x) = 1$.
- Degré 2:
$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = \frac{1}{2}(3x^2 - 1)$$

Interpolation et polynômes II

- **Polynômes orthogonaux**: Il minimise la norme
$$\int (f(x) - P(x))^2 w(x)$$
- on obtient
$$P(x) = \sum_{k=0}^{p} c_k P_k(x) w(x)$$
avec $P_k(x)$ des polynômes de degré $k < p$ donnés par une règle de récurrence, $w(x)$ le poids. Les coefficients c_k sont définis par
$$c_k = \int_a^b f(x) P_k(x) w(x) \approx \sum_{j=0}^{n-1} (x_{j+1} - x_j) \frac{g(x_{j+1}) + g(x_j)}{2}$$
avec $g(x) = f(x) P_k(x) w(x)$ et n un nombre de points.
- On applique une formule du trapèze à chaque intervalle $[x_{j+1} - x_j]$
- Exemple: polynôme de Tchebychev (poids $w(x) = (1 - x^2)^{-\frac{1}{2}}$. Degré 2:
$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = 2x^2 - 1$$

Classe dérivées de polynômes

- On considère les polynômes $\mathbb{K}[X]$ à valeurs dans $\mathbb{K}_2$.
- On avait utilisé :

```
1 template <class T, class X> class polynome
```

- T joue le rôle de $\mathbb{K}$ (entier, rationnel, double, complexe, etc) et X le rôle de $\mathbb{K}_2$ (double, complexe).
- Ici On se restreint pour l'interpolation à $\mathbb{K} = \mathbb{K}_2$ donc $T = X$.
- On va construire des classes dérivées: une pour les polynômes de lagrange, une autre pour les polynômes orthogonaux.
- On veut donc que nos classes héritent de :

```
1 polynome<X,X>
```

- et non de

```
1 polynome<T,X>
```

- On veut donc hériter d'une spécialisation de la classe ”polynome”.

Héritage et template: principe

- Les templates sont compatibles avec l'héritage. Pas de raison que ce ne soit pas le cas.
- Différents cas possibles:
  - Cas 1: Classe "ordinaire" qui dérive d'une template de classe:

```
1 class fille : public mere <type>
```

  - Cas 2: Template de classe qui dérive d'une classe "ordinaire:

```
1 template <class type> class fille : public mere
```

  - Cas 3: Template de classe qui dérive d'un template de classe (même template):

```
1 template <class type> class fille : public mere<type>
```

  - Cas 4: Template de classe qui dérive d'un template de classe:

```
1 template <class type1, class type2> class fille : public mere<type2>
```

Classe Lagrange: constructeurs I

Remarque

- **Pour commencer**: les membres de ”polynôme” sont ”protected”.

- Pour construire les polynômes de Lagrange: on a besoin (en plus) de n points: $x0, ...x_{n-1}$ et n évaluation de la fonction: $f(x_0), ...f(x_{n-1})$.
- On obtient donc

```
template <class X > class lagrange : public polynome<X,X> {
protected:
  int np; // nombre de point
  X * points;
  X * fpoints;
public:
  lagrange(): polynome<X,X>(){
    np = 0;
    points = NULL;
    fpoints = NULL;
  }
....
```

- On hérite de polynome$< X, X >$ donc on un template à un paramètre.

Classe Lagrange: constructeurs II

- **Remarque**: dans une classe héritée d'une classe template il faut utiliser this pour les membres de la classe mère.
- Données de polynômes sont protégées. Accès dans les constructeurs.
- Constructeur par le nombres de points.

```
1  lagrange(int nbpoint){
2      np = nbpoint;
3      this->deg = nbpoint-1;
4      this->a = 0.0; this->b = 0.0;
5      points = new X[this->np];
6      fpoints = new X[this->np];
7      for(int i =0; i<this->np;i++){
8        points[i] = 0.0;  fpoints[i] = 0.0; }
9      this->coefs = new X[this->deg+1];
10     for(int i =0; i<this->deg+1;i++){
11       this->coefs[i] = X();   }
12  }
```

- **Remarque**: le degré est toujours égal $n - 1$. On impose cela par les constructeurs.
- Un utilisateur ne pourra pas créer un polynôme de Lagrange de degré différent que $n - 1$.

Classe Lagrange: constructeurs II

- **Remarque**: dans une classe héritée d'une classe template il faut utiliser this pour les membres de la classe mère.
- Données de polynômes sont protégées. Accès dans les constructeurs.
- Constructeur par copie.

```
lagrange(const lagrange & x): polynome<X,X> (x) {
  np = x.np;
  if(x.points != NULL){
    points = new X[np];
    fpoints = new X[np];
    for(int i =0; i<np;i++){
      points[i] = x.points[i];
      fpoints[i] = x.points[i]; }
  }
}
```

- **Remarque**: le degré est toujours égal $n-1$. On impose cela par les constructeurs.
- Un utilisateur ne pourra pas créer un polynôme de Lagrange de degré différent que $n-1$.

Classe Lagrange: destructeur/méthodes

- Le Destructeur:

```
1 ~lagrange (){
2     if(points != NULL){
3       delete [] points;  }
4     if(fpoints != NULL){
5       delete [] fpoints; }
6   }
```

- Le destructeur de Lagrange appelle à la fin le destructeur de la classe mère.
- Ensuite on ajoute des méthodes pour remplir "points" et fpoints":

```
1 int add_fpoints(X * pf, int n);
2 template <class X > int lagrange<X>:: add_fpoints( X * pf, int n){
3   if(n != np){cout<<" Le nombre de points ... "<<endl; exit(0); }
4   if(pf != NULL){
5     for(int i =0; i<np;i++){
6       fpoints[i] = pf[i];  }
7   }
8   return 0;
9 }
```

Classe Lagrange: méthodes 1

- On veut calculer le polynôme:

$$L_i(x) = \prod_{i=0, i\neq j} \frac{x - x_i}{x_j - x_i}$$

- On implémente uniquement le cas $n = 3$.

```
template <class X> polynome<X,X> lagrange<X>:: lagrange_k(int i) {
  polynome<X,X> m1(1), m2(1);
  polynome<X,X> res(2);
  int im, ip;
  X a,b;
  if(np != 3){ cout<<" np!=3 (degree 2) non implémenté"<<endl; exit(0);}
  else {
    ip = (i+1) % 3; im = (i+2) % 3;
    a = - points[im]/(points[i]-points[im]);
    b = 1.0/(points[i]-points[im]);
    m1.set(a,0);  m1.set(b,1);
    a = - points[ip]/(points[i]-points[ip]);
    b = 1.0/(points[i]-points[ip]);
    m2.set(a,0);   m2.set(b,1);
    res= m1*m2; }
  return res;
}
```

Classe Lagrange: méthodes 2

- On veut calculer le polynôme:

$$P(x) = \sum_{i=0}^{n-1} f(x_i) L_i(x)$$

- On implémente uniquement le cas $n = 3$.

```
template <class X> lagrange<X> lagrange<X>:: construct() {
  if(np != 3){ cout<<" np!=3 (degree 2) non implémenté"<<endl; exit(0);}
  else {
    polynome<X,X> lag(2);
    polynome<X,X> res(2);
    double a;
    for(int i=0; i<np; i++){
      lag=this->lagrange_k(i);
      res= fpoints[i] * lag + res;
    }
    for(int i=0; i<np; i++){
      a =res.get(i);
      this->set(a,i); }
    return *this;
  }
}
```

Classe Lagrange: test unitaire

- Test unitaire:

```
template <class X> int lagrange<X>:: testu(){
  double points[3]={-0.1,0.0,0.1};
  double fm, f0, fp, fref;
  fm = (-0.1)*(-0.1)+3.0*(-0.1)+1.0;
  f0 = (0.0)*(0.0)+3.0*(0.0)+1.0;
  fp = (0.1)*(0.1)+3.0*(0.1)+1.0;
  fref=  (-0.027)*(-0.027)+3.0*(-0.027)+1.0;

  double fpoints[3]={fm,f0,fp};
  lagrange<double> p(3);
  *this = p;
  this->add_points(points,3);
  this->add_fpoints(fpoints,3);
  this->construct();

  if( abs(fref-(*this)(-0.027))< 0.000000001 ) {
    cout<<"Tests unitaires réussis, classe lagrange: 1/1"<<endl;
  } else  {
    cout<<"Tests unitaires réussis, classe lagrange: 0/1"<<endl;
  }
  return 0;
}
```

Classe Orthogonaux: constructeurs I

- Pour construire les polynômes orthogonaux: on a besoin (en plus) de n points: $x0, ...x_{n-1}$ et n évaluations de la fonction : $f(x_0), ...f(x_{n-1})$ pour l'intégration numérique des coefficients.
- On a des méthodes en plus pour "le poids" et "les polynomes" (Legendre, Tchebychev). On détermine le type de polynôme par un nombre.

```
template <class X > class orthogonaux : public polynome<X,X> {
protected:
  int np; // nombre de point
  X * points;
  X * fpoints;
  int type_polynome;
public:
  orthogonaux(): polynome<X,X>(){
    np = 0;
    points = NULL;
    type_polynome =0;
  }
....
```

- On aurait pu faire une classe "orthogonaux" et des classes dérivées pour chaque type de polynômes.

Classe Orthogonaux: constructeurs II

- Exemple constructeur:

```
orthogonaux(int nbpoint,int degree, int type){
   np = nbpoint;
   type_polynome = type;
   this->deg =degree;
   this->a = 0.0; this->b = 0.0;
   points = new X[this->np];
   fpoints = new X[this->np];
   for(int i =0; i<this->np;i++){
     points[i] = 0.0;
     fpoints[i] = 0.0;
   }
   this->coefs = new X[this->deg+1];
   for(int i =0; i<this->deg+1;i++){
     this->coefs[i] = X();}
 }
```

- De la même façon que pour les polynômes de Lagrange, on ajoute des fonctions pour modifier les points et construire les polynômes.

Classe Orthogonaux: III

- Construction du polynomes:

```
1  template <class  X> orthogonaux<X> orthogonaux<X>::construct(){
2    polynome<X,X> res;
3    X coefk=X(), a=X();
4
5    if(this->deg > 2){ cout<<" les cas < au degree 2 ne sont pas implémentés"<<endl
6    else {
7      res = polynome<X,X>(2);
8      for(int i=0; i<this->deg+1; i++){
9        polynome<X,X> p(i);
10       p = this->orthogonaux_k(i);
11       coefk = this->integral(p);
12       res= 0.5*(2*i+1) * coefk * p + res;
13     }
14     for(int i=0; i<this->deg+1; i++){
15       a =res.get(i);
16       this->set(a,i); }
17   }
18   return *this;
19 }
```

- On remarque les fonctions ”get” et ”set” qui sont héritées de la classe mère ”polynôme” peuvent être utilisées sans problème.

Classe Orthogonaux: IV

- La fonction ”orthogonaux_k” calcul le k eme polynôme de Legendre ou autre. La fonction ”integral” calcul

$$\int f(x)p_k(x) \approx \sum_k \frac{1}{2}(x_{k_1} - x_k)(f(x_k) + f(x_{k+1}))$$

```
template <class X> X orthogonaux<X>::integral(polynome<X,X> & p){
  X res = X();
  X xg=X(), xd=X(), fg=X(), fd=X(), loc=X();
  X loc1=X(), loc2=X(), pg=X(), pd=X();
  for(int j=0;j<np-1;j++){
    xg = points[j ];
    xd = points[j+1];
    fg = fpoints[j];
    fd = fpoints[j+1];
    pg = p(xg);  pd = p(xd);
    loc1 = (fg*pg)*this->poids(xg);
    loc2 = (fd*pd)*this->poids(xd);
    loc = 0.5*abs(xd-xg)*(loc1+loc2);
    res = res + loc;
  }
  return res;
}
```

Conclusion

Application de l'héritage

- Il existe un nombre important de types de polynômes: Lagrange, Legendre, Laguerre, Hermite, Splines. Leur construction est différente et ils peuvent utiliser des données différentes.
- En créant une classe par type de polynômes, on peut spécifier la construction et les données nécessaires à cette construction.
- En héritant de la classe générale ils peuvent tous utiliser l'arithmétique et le calcul différentiel des polynômes qui est commun à chaque type de polynôme.
- On peut implicitement flécher/interdire des comportements non prévu ou faux: exemple le ”+” complexe.