Programmation avancée en C++: Templates II

Emmanuel Franck

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

1. Généralités

Rappel sur les templates

Template de fonction

- Cela permet de construire une famille de fonctions qui sont paramétrées par une classe (un type classique ou complexe) ou même un entier.
- Cela nécessite une implémentation valide pour toutes les classes avec lequel cela peut être utilisé.
- On peut quand même spécifier une implémentation pour une classe donnée.

Template de classe

- Cela permet de construire une classe red)paramétrée par une classe(un type classique ou complexe).
- Cela nécessite une implémentation valide pour toutes les classes avec lequel cela peut être utilisé mais comme sur les fonctions on peut spécifier une implémentation pour une classe donnée.
- On une classe qui dépend d'un template T. Pout chaque T on obtient une classe différente. *complexe* < *int* > et *complexe* < *double* > sont deux classes différentes sans opérateur caste associé.

Arithmétique: anneaux/corps quadratiques

- En arithmétiques on peut introduire l'anneau $\mathbb{Z}(d)$ ou le corps $\mathbb{Q}(d)$ défini par

$$\mathbb{Z}(d) = \left\{a + \sqrt{d}b, \text{ with } (a,b) \in \mathbb{Z}\right\}$$

$$\mathbb{Q}(d) = \left\{a + \sqrt{d}b, \text{ with } (a,b) \in \mathbb{Q}\right\}$$

  et

$$x^2 - d = 0$$

- Dans l'exemple on considère $d > 0 \in \mathbb{N}$. Si on prend $\alpha, \beta \in \mathbb{Q}(d)$ la somme et le produit des deux sont dans le même ensemble.
- On considère $\alpha \in \mathbb{Q}(3)$, $\beta \in \mathbb{Q}(2)$, La somme des deux n'est pas dans $\mathbb{Q}(d)$.
- Pour chaque d, l'addition interne, la multiplication interne sont différentes. On souhaite donc une classe pour chaque d.
- **Solution**: comme pour les fonctions on peut faire un template de classe paramétré par un "entier" (pas possible avec un double). On utilisera une intégration approchée.

2. Template de classe

Definition du template de classe

- On propose une classe ou le type de a et b est paramétré par un template. Cela permet de générer à la fois $\mathbb{Z}(d)$ et $\mathbb{Q}(d)$.

```
1  template <class T, int d> class anneauquad{
2    T a;
3    T b;
4  public:
5    anneauquad (){
6      a = T();  b = T();
7    }
8    anneauquad (T ainit , T binit){
9      a = ainit; b = binit;
10   }
11   anneauquad (const anneauquad & x){
12     a = x.a;  b = x.b;
13   }
14   ~anneauquad (){
15   }
16 ......
```

- On crée des templates de classe avec des paramètres de type ”int” de la même façon que pour les templates de fonctions.

Definition du template de classe II

- On propose de définir les opérateurs internes et externes. Ici $\mathbb{Z}(d)$ et $\mathbb{Q}(q)$ sous des sous-groupes de $\mathbb{R}$. Par conséquent on souhaite définir les opérations externes avec les ”double”.

```
1   anneauquad set_a (T);
2   anneauquad set_b (T);
3   anneauquad operator + (anneauquad);
4   anneauquad operator - (anneauquad);
5   anneauquad operator * (anneauquad);
6   anneauquad & operator = (const anneauquad &);
7   template <class U, int e>
8   friend double operator + (double, anneauquad<U,e>);
9   template <class U, int e>
10  friend double operator * (double, anneauquad<U,e>);
11  template <class U, int e>
12  friend ostream & operator << (ostream &, const anneauquad<U,e> &);
```

- On crée des templates de classe avec des paramètres de type ”int” de la même façon que pour les templates de fonctions.
- Comme précédemment pour les fonctions externes (amies) on doit préciser qu’il s’agit d’un template ainsi que la forme du template.

Definition des méthodes et opérateurs I

- La construction des méthodes se fait de la même façon que précédemment

```
1  template <class T, int d> anneauquad<T,d> anneauquad<T,d>::set_a (T aa){
2      a = aa;
3      return *this;
4  }
5  template <class T, int d> anneauquad<T,d> anneauquad<T,d>::
6          operator + (anneauquad<T,d> x){
7    anneauquad<T,d> res;
8    res.a = a + x.a;
9    res.b = b + x.b;
10   return res;
11 }
12 template <class T, int d> anneauquad<T,d> anneauquad<T,d>
13         ::operator * (anneauquad<T,d> x){
14   anneauquad<T,d> res;
15
16   res.a = a*x.a + d * (b*x.b);
17   res.b = a*x.b + b*x.a;
18   return res;
19 }
```

Définition des méthodes et opérateurs II

- Pour les opérations externes. Il faut faire un peu plus attention.
- On a a et b dans $\mathbb{Z}$ ou $\mathbb{Q}$. On convertit a et b dans $\mathbb{R}$ puis on fait l'opération.

```
1  template <class T, int d> double operator * (double y, anneauquad<T,d> x){
2    double res;
3
4    res = y*( (double)x.a + sqrt(d) * (double)x.b);
5    return res;
6  }
7
8  template <class T, int d> double operator + (double y, anneauquad<T,d> x){
9    double res;
10
11   res = y+( (double)x.a + sqrt(d) * (double)x.b);
12   return res;
13 }
```

- **Important**: il faut donc que l'opérateur "cast" vers les doubles soit défini pour les types T qu'on utilisera en pratique.

Définition des méthodes et opérateurs III

- On souhaite utiliser la classe pour $\mathbb{Z}$ (les ”int”) et $\mathbb{Q}$ (notre classe ”Rationnel”).
- Dans le premier cas, l'opérateur ”cast” existe. Dans le second il faut l'ajouter à la classe ”Rationnel”.
- Dans la classe ”Rationnel”:

```
1  operator int (); // cast rationnel en int
2  operator double (); // cast rationnel en double
```

```
1  rationnel:: operator int (){
2    int res;
3    res = n/d;
4    return res;
5  }
6  rationnel:: operator double (){
7    double res;
8    res = ((double) n )/((double) d);
9    return res;
10 }
```

- Il y a pas de type de retour dans la déclaration des opérateurs de ”cast”.

Utilisation I

- Le principe d'utilisation c'est similaire au template de classe juste avec des types.
- Exemple de tests unitaires:

```
1  template <class T, int d> int anneauquad<T,d>::testu_q(){
2
3    anneauquad<rationnel,3> x(rationnel(1,1),rationnel(1,2));
4    anneauquad<rationnel,3> y(rationnel(1,1),rationnel(2,1));
5    anneauquad<rationnel,3> z;
6    double p= 1.2;
7
8    *this = x; //((1/1) + (1/2)sqrt(3))
9    y = y + x; // 2 +(5/2) sqrt(3))
10   z = y * x; //23/4 + 7/2 sqrt(3)  = 11.812177
11   p = (p + z) + p*x; // 13.012177 + 2.23923 = 15.251407
12
13   if(p > 15.2514 && p < 15.2515){ return 0; }
14   else{ return 1; }
15 }
```

Utilisation II

- On aurait pu écrire une classe de type

```
1  template <class T> class anneauquad2{
2    T a;
3    T b;
4    int d = 0
5  public:
6    anneauquad2 (){
7      a = T();  b = T();
8    }
9    anneauquad2 (T ainit , T binit){
10     a = ainit; b = binit;
11   }
12    anneauquad2 (T ainit , T binit, int dinit){
13     a = ainit; b = binit; d= dinit;
14   }
15   anneauquad2 (const anneauquad2 & x){
16     a = x.a;  b = x.b; d = x.d;
17   }
18   ~anneauquad2 (){
19   }
20 ......
```

- On va comparer les deux solutions.

Utilisation III

- Utilisation de classe sans template de ”nombre”.

```
1  anneauquad2<int> a(1,2,2); // 1 + 2 sqrt(2)
2  anneauquad2<int> b(1,3,2); // 1 + 3 sqrt(2)
3  anneauquad2<int> c(0,0,2); // 1 + 2 sqrt(2)
4  anneauquad2<int> d(1,3,3); // 1 + 3 sqrt(3)
5  anneauquad2<int> e;
6
7  c = a + b;
8  cout <<" valeur c"<<c<<endl;
9  e = c + d;
```

```
user $  valeur c2 + sqrt(2) 5
user $ les nombres "d" ne sont pas égaux, l'addition␣interne␣n'est pas
     valide.
```

- Les opérations doivent être testées et on doit vérifier si elles sont légales.
- **Problème**: Les objets ”c” et ”d” sont dans la même classe mais pas dans le même ensemble algébrique.

Utilisation III

- Utilisation de classe avec template de ”nombre”.

```
1  anneauquad<int,2> an(1,2); // 1 + 2 sqrt(2)
2  anneauquad<int,2> bn(1,3); // 1 + 3 sqrt(2)
3  anneauquad<int,2> cn(0,0); // 1 + 2 sqrt(2)
4  anneauquad<int,3> dn(1,3); // 1 + 3 sqrt(3)
5  anneauquad<int,3> en;
6
7  cn = an + bn;
8  cout <<" valeur cn"<<c<<endl;
9  en = cn + dn;
```

```
user $ invalid operands to binary expression ('anneauquad<int, 2>' and
     'anneauquad<int, 3>')
user $   en = cn + dn;
```

- Les opérations entre deux espaces $\mathbb{Z}(2)$ et $\mathbb{Z}(3)$ est naturellement illégal.
- **Avantage**: Les objets ”cn” et ”dn” ne sont pas dans la même classe et dans le même ensemble algébrique.
- La structure informatique respecte la structure algébrique.

Utilisation IV

- Comme préciser. A chaque valeur de T (aussi de n) on génère une classe différente à chaque fois.

```
1  anneauquad<int,2> an2(1,2);
2  anneauquad<rationnel,2> bn2(rationnel(1,3),rationnel(1,2));
3  anneauquad<rationnel,2> cn2;
4  cn2 = an2 + bn2;
```

```
user $  invalid operands to binary expression
user $      ('anneauquad<int,␣2>' and 'anneauquad<rationnel,␣2>')
user $  cn2 = an2 + bn2;
```

- Pour $T = rationnel$ ou $T = int$ on obtient .
- C'est aussi vrai pour les classes précédentes: "complexe<>" ,"polynome<, >" etc.

Cast

- On a vu que pour chaque valeur de ”T” et ”d” on obtient une classe différente.
- On peut donc construire des ”cast” entre les différentes classes à l'aide de la spécialisation.

```
operator anneauquad<rationnel,2>();
operator anneauquad<int,2>();
template<>
anneauquad<int,2>::operator anneauquad<rationnel,2>(){
  anneauquad<rationnel,2> x;
  x.set_a(rationnel(a,1));
  x.set_b(rationnel(b,1));
  return x; }
template<>
anneauquad<long int,2>::operator anneauquad<rationnel,2>(){
  anneauquad<rationnel,2> x;
  x.set_a(rationnel(a,1));
  x.set_b(rationnel(b,1));
  return x; }
template<>
anneauquad<rationnel,2>::operator anneauquad<int,2>(){
  anneauquad<int,2> x;
  x.set_a( (int) (a.num()/a.dem()));
  x.set_b( (int) (b.num()/b.dem()));
  return x; }
```

Cast

```
1  operator anneauquad<rationnel,2>();
2  operator anneauquad<int,2>();
3  template<>
4  anneauquad<int,2>::operator anneauquad<rationnel,2>(){
5    anneauquad<rationnel,2> x;
6    x.set_a(rationnel(a,1));
7    x.set_b(rationnel(b,1));
8    return x; }
9  template<>
10 anneauquad<long int,2>::operator anneauquad<rationnel,2>(){
11   anneauquad<rationnel,2> x;
12   x.set_a(rationnel(a,1));
13   x.set_b(rationnel(b,1));
14   return x; }
15 template<>
16 anneauquad<rationnel,2>::operator anneauquad<int,2>(){
17   anneauquad<int,2> x;
18   x.set_a( (int) (a.num()/a.dem()));
19   x.set_b( (int) (b.num()/b.dem()));
20   return x; }
```

- La première fonction permet de convertir en ”anneauquad rationnel”. Elle appartient donc au cas ”int ou ”long” int.
- La 2ème fonction appartient au cas ”rationnel”.

Valeur défaut d'un template

- On peut spécifier des valeurs par défaut aux paramètres de template.
- Exemple 1:

```
1  template <class T, int d=1> class anneauquad{...}
```

- On peut aussi faire:

```
1  template <class T=int, int d=1> class anneauquad{...}
2  template <class T=rationnel, int d=1> class anneauquad{...}
```

- **Cas interdit**:

```
1  template <class T=int, int d> class anneauquad{...}
```

- **Explication**: Lorsqu'on donne une valeur par défaut à un paramètre, on doit donner des valeurs par défaut à tous les paramètres qui suivent.
- Utilisation:

Conclusion

- **Remarque**: on peut aussi spécialiser les templates par un entier.

```
1 template <>
2 anneauquad<int,0> anneauquad<int,0>::set_b (int bb){
3   cout<<"Puisque d est nul la valeur de b est arbitraire"<<endl;
4   return *this;
5 }
```

- Les classes comme les fonctions, peuvent avoir des templates dépendant de paramètres.
- Les déclarations et définitions marchent de la même façon.
- Par contre on ne peut le faire que pour certains types de paramètres. En pratique, le ”int” et le ”char” sont les plus utilisés. Impossible avec les ”double”.
- Comme démontré précédemment cela peut permettre de générer des classes robustes proches de certaines structures mathématiques.

3. Autre exemple : Espace vectoriel des polynômes

Espace vectoriel normé des polynômes I

- Comme vu précédemment l'espace de polynômes $\mathbb{R}[X]$ est un espace vectoriel normé. On peut le définir sur un segment $[a, b]$.
- Dans ce cas-là on peut souhaiter ne pas mixer des polynômes avec différents segments. En effet comment décrire l'intégration dans ces cas etc ?
- **Possibilité**: tester à chaque fois que les segments sont égaux (traitement du problème au niveau de l'exécution pas de la compilation).
- **Autre possibilité**: les templates.
- On aurait envie de faire

```
1 template< double a ,double b> class polynome
```

- Avec des entiers, on a le droit. Pas avec des "double".
- En effet, l'égalité stricte entre deux "double" est impossible. On doit pouvoir tester l'égalité parfaite.

Espace vectoriel normé des polynômes II

- **Question**: que peut t'on utiliser comme paramètre d'un template.
- Possibilité:
  - classe ou type classique,
  - **type intégral**: char, int, long, short et leurs versions signées et non signées,
  - pointeur ou référence d'objet,
  - pointeur ou référence de fonction.
- **Pourquoi**: Pour les lettres (”char”), les entiers, les adresses (pointeurs) on peut tester une égalité stricte.

Espace vectoriel normé des polynômes III

- Implémentation pratique:

```
1  class segment {
2  public:
3    static constexpr double a = 0.0;
4    static constexpr double b = 1.0;
5  };
6  template <class segment_type> class polynome {
7    int deg;
8    complexe * coefs;
9    ....
10 }
11 template <class segment_type> complexe polynome<segment_type>::integral(){
12   complexe res(0.0,0.0);
13   polynome<segment_type> primitive;
14   primitive = this->primitive();
15
16   res= primitive(complexe(segment_type::b,0.0))
17   -primitive(complexe(segment_type::a,0.0));
18   return res;
19 }
```

Espace vectoriel normé des polynômes IV

- On crée une classe segment puis on crée un template de classe et donc pour chaque segment on a une classe.
- Si on ne donne pas de segment on utilise le ”segment_ref”:

```
1 template <class segment_type=segment_ref> class polynome{...}
```

- Implémentation de la classe segment:
  - constexpr: on ne peut pas modifier ”a” et ”b”. Permets d’exprimer des expressions ”constexpr”, au sens où le compilateur les évaluera à la compilation si cela s’avère possible.
  - static: a et b sont définis pour tous les objets de classe et on n’a pas besoin de les instancier avec un constructeur.
- Si votre classe segment ne contient pas ”a”, ”b”: erreur. En effet:

```
1 segment_type::b
```

- Cette ligne utilisée dans la fonction précédente, suppose que la classe ”segment_type” contient un membre public *b* (idem pour a).

EVN des polynômes orthogonaux

- **Polynômes orthogonaux**: Legendre, Hermite, Laguerre, Tchebychev ...
- Principe:

$$f(x) \approx \sum_{i}^{n} \alpha_i P_i(x)$$

avec

$$\alpha_i = \frac{(f, P_i(x))}{(P_i, P_i)}$$

et

$$(f, g) = \int_a^b f(x)g(x)W(x)dx$$

avec $W(x)$ un poids. On a un espace vectoriel par segment (comme avant) et par poids $W(x)$.

- Exemple: Legendre $W(x) = 1$, Hermite $W(x) = e^{-x^2}$ etc.
- Représentation très utilisée pour les fonctions et les équations physiques.
- On peut gérer ce genre d'espace vectoriel/ de représentation avec les templates.

EVN des polynômes orthogonaux II

- **Idée**: on veut paramétrer une classe par une fonction.

```
1 template <double (*weight)(double), class segment_type=segment_ref>
2 class polynome {
3   int deg;
4   complexe * coefs;
5   public:
6   ....}
```

- ”weight” est un pointeur de fonction (une fonction prenant un double et envoyant un double) qui paramètrise la classe.
- A chaque fonction donnée cela génère une classe différente.
- Utilisation:

```
1 double legendre(double x){
2   return 1.0; }
3 double hermite(double x){
4   return exp(-x*x); }
5 int main (){
6   polynome<hermite,segment> A(1);
7   polynome<legendre,segment> B(1);
8   ...
```

EVN des polynômes orthogonaux III

- Implémentation du produit scalaire:

$$\int_a^b P(x)Q(x)W(x)dx \approx (b-a)\left(\frac{P(a)Q(a)W(a)+P(b)Q(b)W(b)}{2}\right)$$

- Intégration numérique très simple. En pratique on utiliserait une intégration plus précise.

```
1  template <double (*weight)(double),class segment_type>
2    complexe polynome<weight,segment_type>::integral(){
3    complexe res(0.0,0.0);
4    complexe lenght(0.0,0.0);
5    complexe ea, eb;
6    lenght =complexe(segment_type::b,0.0)-complexe(segment_type::a,0.0);
7    ea = complexe(weight(segment_type::a),0.0);
8    eb = complexe(weight(segment_type::b),0.0);
9    res=(*this)(complexe(segment_type::b,0.0))*eb
10         +(*this)(complexe(segment_type::a,0.0))*ea;
11   return res;
12 }
```

Utilisation du mot clé ”auto”

- L’utilisation de template ou de template de template complexifie vite les notations.
- On rappelle: ”auto” permet de déclarer une variable par son initialisation.
- Cas classique:

```
1 template <class segment_type> complexe polynome<segment_type>::integral(){
2   complexe res(0.0,0.0);
3   polynome<segment_type> primitive;
4   primitive = this->primitive();
5   ....
6 }
```

```
1 template <class segment_type> complexe polynome<segment_type>::integral(){
2   complexe res(0.0,0.0);
3   auto primitive= (*this);
4   primitive = this->primitive();
5   ....
6 }
```

Conclusion

- Template: Outils très puissant mais difficile.
  - Emboîter des templates peut vite complexifier un code,
  - il faut écrire des fonctions génériques pour toutes valeurs du paramètre de template. Exemple que ”==” ou ”¡” soit défini pour les classes utilisées comme paramètres.
  - On peut pas paramétriser par n’importe quoi. Pas toujours évident.
  - La spécialisation est assez limitée.
  - Dans le C++ récent, des concepts plus évolués. Exemple: template de variable.

```
1  template<class T>
2  constexpr T pi = T(3.14159265358979323851);
3  cout<<"cas 1>>>"<<pi<int><<endl;
4  cout<<"cas 2>>>"<<pi<double><<endl;
```

```
user $ cas 1>>>3
user $ cas 2>>>3.14159
```