Programmation générique [mo]
Support de cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 30 mai 2018

Table des matières

C++ - Programmation générique

Mots-Clés Modèles ■
Requis Axiomatique impérative, Axiomatique objet ■
Difficulté • ○ ○ (1 h) ■

Introduction
Ce module décrit la **programmation générique** appelée aussi polymorphisme paramétrique.

1 Programmation générique

Programmation générique
(*generic programming* en anglais) Permet de décrire des comportements identiques, indépendamment du type de données. On parle aussi de **polymorphisme paramétrique**.

Remarque
De tels modèles de traitements/classes **génériques** s'appellent aussi **traitements/-classes génériques** ou **patrons** (chablons) ou encore modèles (*template* an anglais).

Exemple : Modèle de fonctions
L'algorithme `max(x,y)` est un modèle de fonctions (et non une fonction) : c'est le même modèle paramétré par un type `T` qui dispose de l'opérateur de comparaison `<`. De même, l'algorithme `swap(x,y)` est un modèle de procédures : c'est la même procédure paramétré par un type `T` des paramètres de la procédure.

Exemple : Modèle de classes
En C++, le `vector` est un modèle de classes (et non une classe) : c'est le même modèle que l'on y stocke des `char` (`vector<char>`), des `int` (`vector<int>`) ou tout autre objet.

Conclusion
Les modèles de modules/classes sont un **moyen condensé** d'écrire plein de modules/-classes potentiels à la fois. L'apport des « génériques » permet également de rendre le code plus robuste et simplifie grandement la programmation.

2 Modèle de modules

2.1 Mise en oeuvre d'un modèle procédural

Définition d'un modèle procédural

La définition des modèles de procédure/fonction ne génère en elle-même aucun code : c'est une description de codes potentiels.

Définition d'un modèle procédural

```
template<typename T, ... int N>
//... ici figure la déclaration/définition d'un module
//... où T apparait comme type et N comme constante
```

Explication

Déclare une famille de modules.

Instanciation d'un modèle procédural

Le code n'est produit que lorsque **tous les paramètres du modèle ont un type spécifique**, c.-à-d. qu'il faut fournir des valeurs pour tous les paramètres (au moins ceux qui n'ont pas de valeur par défaut). On appelle cette opération, une **instanciation du modèle**.

Instanciation implicite

Dans le cas des modèles de procédure/fonction, l'**instanciation** peut être **implicite** lorsque le **contexte** permet au compilateur de décider de l'instance de modèle à choisir.

Instanciation explicite

L'**instanciation explicite** peut être utile dans les cas où le contexte (dans les modèles de procédure/fonction) n'est pas suffisamment clair pour choisir.

2.2 Exemple : Modèle de fonctions

Le modèle de déclaration d'une fonction `max` typique est :

```
template<typename T>
T max(const T& a,const T& b)
{
   return (a < b ? b : a);
}
```

La fonction `max` est ensuite instanciée automatiquement lors de ses appels, par exemple :

```
//Instanciation automatique de la fonction template max
int main(int, char **)
{
  int i, j;
  double a, b;
  ...
  cout << max(i,j) << endl;
  cout << max(a,b) << endl;
  return 0;
}
```

La ligne `cout<<max(i,j)<<endl;` créée la version :

```
int max(const int&, const int&)
```

à partir du `template|, alors que \lstinline`coutmax(a,b)endl ;@ créée la version :

```
double max(const double&, const double&)
```

Mais il n'y a pas de conversion automatiques entre les types. Ainsi l'instruction suivante est ambigüe :

```
cout<<max(i,b)<<endl;
```

Il faut écrire une instanciation explicite afin d'avoir un type `T` identique. Par exemple pour un `double` :

```
cout<<max(double(i) b); // première solution
cout<<max<double>(i,b); // autre solution
```

2.3 Exemple : Modèle de procédures

Le modèle de procédures `permuter` lequel permute le contenu de ses deux paramètres s'écrit :

```
template<typename T>
void permuter(T& a, T& b)
{
  T tmp(a);
  a = b;
  b = tmp;
}
```

On peut alors utiliser ce modèle avec tout type/classe pour lequel le constructeur de recopie et l'opérateur d'affectation (=) sont définis.

```
int i1(2), i2(4);
permuter(i1, i2);
```

Par le contexte, il est clair qu'il s'agit de l'instance `permuter<int>`. De même, le code suivant est licite :

```
vector<double> v1, v2;
permuter(v1, v2);
```

Tout comme celui-ci :

```
string s1("ca marche"), s2("coucou");
permuter(s1,s2);
```

3 Modèle de classes

3.1 Mise en oeuvre d'un modèle de classes

Déclaration d'un modèle de classes

```
template<typename T1, ...>
class K { ... };
```

Explication

Déclare le modèle de classe `K` de paramètres (=noms génériques) de type `Ti`. Les types `Ti` (paramètres du modèle) peuvent être utilisés dans la définition qui suit comme tout autre type.

C++ : Mot-clé typename

Ce nouveau mot `typename` a été ajouté par la norme afin de souligner tout-type. Il remplace le mot-clé `class` (des anciennes versions du C++)

Valeur par défaut d'un modèle de classe

Il est possible de définir des types par défaut avec la même contrainte que pour les paramètres de fonction/procédure : les valeurs par défaut doivent être placées en dernier.

Instanciation d'un modèle de classes

Contrairement aux modèles de fonctions/procédures, l'instanciation des classes n'est pas automatique. Il faut la faire explicitement.

3.2 Définition externe des méthodes

Définition externe des méthodes

```
template<typename T1, ...>
K<T2,...>::methode(...)
{
  ...
}
```

Explication

Définit la `methode` du modèle de classes `K`.

Attention

Il est **absolument nécessaire d'ajouter les paramètres du modèle** (les types génériques) au nom de la classe pour bien spécifier que dans cette définition c'est la classe qui est en modèle et non la méthode.

3.3 Exemples : Modèles de classes

Exemple : Modèle Point

```
template<typename T> class Point { ... } ; // modèle de classe
Point<double> p1 ; // instancie un Point de coordonnées réelles
typedef Point<int> PGauss ; // type Point_De_Gauss
```

Exemple : Modèle Paire

On peut vouloir créer une classe qui réalise une paire d'objets :

```
template<typename T1, typename T2>
class Paire {... };
```

On explicite l'instanciation lors de la déclaration d'un objet. Dans le cas d'un modèle de classes, il suffit de spécifier le(s) type(s) désiré(s) après le nom du modèle du classe entre les chevrons (< et >). Ainsi :

- `Paire<string, double>` crée la classe paire string-double.
- `Paire<char, unsigned>` crée la classe paire char-unsigned.

Un tel modèle de classe existe dans la STL définie dans la bibliothèque `<utility>` : c'est la `std::pair<T1,T2>`.

Exemple : Constructeur de la Paire

```
template<typename T1, typename T2>
Paire<T1,T2>::Paire(const T1& a, const T2& b)
: first(a), second(b)
{}
```

Exemple : Valeur par défaut

```
template<typename T1, typename T2 = unsigned>
class Paire
{
  ...
};
```

Cette instruction permet de déclarer la classe paire « char-unsigned » par `Paire<char>`.

4 Compléments

4.1 Surcharge et spécialisation

Surcharge des modèles de fonctions/procédures

Même principe que les fonctions/procédures usuelles.

Version appelée

Lorsqu'il a le choix entre une version dédiée dont le prototype colle directement à l'appel et l'instanciation d'un modèle, le compilateur choisit systématiquement la **version dédiée**. Ce dernier fait est important à connaître lorsque l'on cherche à générer, par exemple, des opérateurs relationnels manquant.

Exemple : Surcharge

```
// Modèle de procédure
template<typename T>
void afficher(const T& x)
{
  cout<<"Affiche"<<x<<endl;
}

// Surdéfinition pour les pointeurs
template<typename T>
void afficher(const T* x)
{
  afficher<T>(*x);
}
```

Spécialisation

Permet de définir une **version particulière** d'une fonction/procédure ou d'une classe pour un choix spécifique des paramètres du modèle.

Lorsqu'elle est totale, elle consiste en :

- Ajouter le mot-clé `template` devant la définition.
- Nommer explicitement le module/classe spécifié.

Exemple : Spécialisation

On peut spécialiser le deuxième modèle (ci-avant) dans le cas des pointeurs sur des entiers [c'est le `<int>` après `afficher`] :

```
template<>
void afficher<int>(const T* x)
{
  cout<<"Affiche le contenu d'un entier: "<<*x<<endl;
}
```

Remarque
La spécialisation :

- Peut s'appliquer à une méthode d'un modèle de classe sans que l'on soit obligé de spécialiser toute la classe. Utilisée de cette façon, elle peut s'avérer particulièrement utile.
- **N'est pas une surcharge** car il n'y a pas génération de plusieurs fonctions de même nom mais bien une **instance spécifique** du modèle.
- Peut être **partielle** (de classes ou de fonctions/procédures).