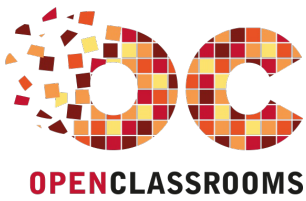
[C++] Les templates

Par foester



www.openclassrooms.com

Sommaire

[C++] Les templates

Par  foester

Mise à jour : 01/01/1970

Un patron, au sens mathématique du terme est la projection planaire du "dépliage" d'un objet en 3 dimensions (appelé polyèdre). On peut donc l'apparenter à un modèle sur lequel on peut se baser afin de construire un objet.

En C++, un patron de fonction (encore appelé ***template***), est le modèle d'une fonction, dont le type de retour et le type des arguments n'est pas fixé.

J'utiliserais ici le terme de patron de fonction, mais ne perdez pas de vue que ce sont les templates!

C'est une alternative à la surcharge (ou surdéfinition) de fonction, qui permet donc de gagner en performances, en temps de codage, et surtout en clarté.

Pour suivre ce tutoriel, vous devez:
> Avoir lu le tutoriel de C++ de M@teo21 (au moins la partie 1)
> Avoir bien compris le fonctionnement de la surcharge de fonctions
Sommaire du tutoriel :

Pour bien comprendre...

Afin de bien vous faire comprendre, nous allons baser ce tutoriel sur un exemple très simple.
Ecrivons une fonction qui nous permettra de calculer la somme de deux nombres (de type ***int***), puis de renvoyer la valeur du résultat.

Pour faire simple (car ce n'est pas très compliqué quand même), voici comment on pourrait écrire cette fonction:

Code : C++

```
int calculerSomme(int operande1, int operande2)
{
    int resultat = operande1 + operande2;
    return resultat;
}
```

Cette fonction marche très bien (chez moi du moins), mais pose le problème suivant: les deux valeurs envoyées en arguments a la fonction doivent être de type ***int***

En réalité, la conversion de type rend l'envoi de variables ***double*** possible.
Une conversion ***double***->***int***, bien que dégradante serait automatiquement effectuée.

Donc si je veux ajouter 2.5 à 3.2, cette fonction ne convient pas!
Heureusement, grâce à M@teo21, nous savons surcharger cette fonction, pour nous permettre d'ajouter non plus des ***int*** mais des ***double***.

La fonction surchargée pour les ***double*** serait:

Code : C++

```
double calculerSomme(double operande1, double operande2)
{
    double resultat = operande1 + operande2;
    return resultat;
}
```

Il en irait évidemment de même pour tous les types (***float***, ***long***, ...).

Créer un patron

Definir le modèle

Dans l'exemple précédent, on peut facilement se rendre compte que les fonctions sont formées sur le même modèle.
Ce modèle ressemble à ceci:

Code : C++

```
type calculerSomme(type operande1, type operande2)
{
    type resultat = operande1 + operande2;
    return resultat;
}
```

C'est cette ressemblance, qui va nous permettre de créer un patron de fonction.

Le patron

Déclarer un patron

Nous allons désormais déclarer notre patron de fonction, afin de remplacer la fonction ***calculerSomme***, sous toutes ses formes (surchargées ou non).

Code : C++

```
template<class Type> Type calculerSomme(Type operande1, Type
operande2)
{
    Type resultat = operande1 + operande2;
    return resultat;
}
```

La norme prévoit que l'on puisse utiliser le mot-clé ***typename*** au lieu de ***class*** dans la déclaration de notre patron.

Cela évite les confusions .

Utiliser le patron

Maintenant que notre patron est déclaré, nous pouvons l'utiliser, comme une fonction classique.

operande1 et ***operande2*** n'ont pas un type défini.
Il doit cependant être le même pour les deux.
On ne peut donc pas envoyer un ***int*** et un ***float*** à notre fonction.

Code : C++

```
#include <iostream>

using namespace std;

// On utilise typename et non pas classe pour eviter les confusions
template<typename Type> Type calculerSomme(Type operande1, Type
operande2)
{
    Type resultat = operande1 + operande2;
    return resultat;
}

int main()
{
 int n = 4, p = 12, q = 0;
        // On appelle notre fonction calculerSomme comme une
fonction normale
 q = calculerSomme(n, p);
 cout << "Le resultat est : " << q << endl;

 return 0;
}
```

Ce code fonctionne très bien et donne comme résultat:

Code : Console

```
Le résultat est : 16
```

L'avantage désormais et que vous pouvez calculer la somme de deux ***float***, en remplaçant juste ceci:

Code : C++

```
int n = 4, p = 12, q = 0;
q = calculerSomme(n, p);
```

par cela:

Code : C++

```
float n = 4.2, p = 12.8, q = 0;
q = calculerSomme(n, p);
```

Les exceptions

On pourrait penser que ce code fonctionnerait:

Code : C++

```
#include <iostream>

using namespace std;

// On utilise typename et non pas classe pour eviter les confusions
template<typename Type> Type calculerSomme(Type operande1, Type
operande2)
{
    Type resultat = operande1 + operande2;
    return resultat;
}

int main()
{
 int n = 4, q = 0;
        unsigned int p = 12;
 q = calculerSomme(n, p);
 cout << "Le resultat est : " << q << endl;

 return 0;
}
```

Mais il ne fonctionne pas. Ceci est dû au fait que le compilateur considère les type ***unsigned int*** et ***int*** comme étant différents.

Il en va de même pour un type ***const int*** et ***int***.

Ce n'est pas valable uniquement pour les ***int***, mais pour tous les types qui sont concernés par ***unsigned*** et *const*.

Les paramètres

Théorie

Jusqu'à maintenant, nous avons utilisé le même type pour tous les arguments de notre patron.
Or il peut arriver que l'on ai besoin d'envoyer des arguments dont le type ne sera pas le même.

Si l'on reprend l'exemple utilisé jusque là, et qu'on le modifie pour pouvoir additionner un ***int*** avec un ***double***, on obtient ceci:

Code : C++

```
double calculerSomme(int operande1, double operande2)
{
    double resultat = operande1 + operande2;
    return resultat;
}
```

On pourrait le surcharger comme ceci:

Code : C++

```
double calculerSomme(double operande1, int operande2)
{
    double resultat = operande1 + operande2;
    return resultat;
}
```

ou même si l'on avait besoin:

Code : C++

```
float calculerSomme(int operande1, float operande2)
{
    float resultat = operande1 + operande2;
    return resultat;
}
```

Et bien pour cela, on va utiliser plusieurs type d'arguments dans notre patron.
On va faire comme ci (modèle):

Code : C++

```
template<typename Type1, typename Type2> Type2 calculerSomme(Type1
operande1, Type2 operande2)
{
    Type2 resultat = operande1 + operande2;
    return resultat;
}
```

Exemples

Code : C++

```
#include <iostream>
using namespace std ;

template<typename Type1, typename Type2> Type2 calculerSomme(Type1
operande1, Type2 operande2)
{
    Type2 resultat = operande1 + operande2;
    return resultat;
}

int main()
{
 int a = 20;
 float b = 10.2;
 cout << "La somme vaut " << calculerSomme(a, b) << endl;

 system("PAUSE");
 return 0;
}
```

Ce code renvoie 30.2
Vous pouvez bien sur inverser ***a*** et ***b*** lors de l'appel de ***calculerSomme***.

Notre fonction patron renvoie une donnée de type ***Type2***.
Si l'on envoie a en deuxième argument, le type de retour devient donc ***int***.
Ainsi la valeur de ***b*** est transformée en ***int*** (au lieu de ***float***).

Vous pouvez aussi envoyer deux arguments du même type. La déclaration de notre patron n'oblige pas à utiliser 2 types différents, mais le permet simplement . Ainsi le code suivant fonctionne:

Code : C++

```
#include <iostream>
using namespace std ;

template<typename Type1, typename Type2> Type2 calculerSomme(Type1
operande1, Type2 operande2)
{
    Type2 resultat = operande1 + operande2;
    return resultat;
}

int main()
{
 int a = 20;
 int b = 10;
 cout << "La somme vaut " << calculerSomme(b, a) << endl;

 system("PAUSE");
 return 0;
}
```

Surcharger un patron

Les fonctions patrons sont avant tout des fonctions, ce qui induit qu'elles peuvent être surchargées.

Si l'on reprend notre bonne vieille fonction ***calculerSomme***, il nous est possible de la surcharger par exemple comme ceci:

Code : C++

```
int calculerSomme(int operande1, int operande2, int operande3)
{
    int resultat = operande1 + operande2 + operande3;
    return resultat;
}
```

Et bien il est possible d'en faire autant avec les fonctions patrons. Notre fonction patron précédente peut être surchargée ainsi:

Code : C++

```
template<class Type> Type calculerSomme(Type operande1, Type
operande2, Type operande3)
{
    Type resultat = operande1 + operande2 + operande3;
    return resultat;
}
```

Le principe est donc exactement le même que pour une fonction lambda.
Vous ne devriez donc pas être surpris, ni avoir de problème de compréhension ici

Spécialisation

Vous l'avez désormais compris, une fonction patron est une sorte d'ensemble de fonctions, pouvant avoir des arguments différents en nombre et en type.

Par contre, quelque soit le type des arguments, les calculs qui sont réalisés dans la fonction sont identiques. On dira que les algorithmes sont identiques. Cela peut parfois poser des problèmes, dans le cas ou vous voulez changer l'algorithme pour un type spécifique.

Comprendre le compilateur

Lors de l'appel d'une fonction, le compilateur va chercher en premier les fonctions qui correspondent complètement à ce que l'utilisateur demande.
En d'autres termes, si le compilateur rencontre la ligne:

Code : C++

```
calculerSomme(int a, int b);
```

Il va chercher la fonction dont le prototype est de la forme:

Code : C++

```
type calculerSomme(int, int);
```

Il peut arriver que le compilateur ne trouve pas exactement ce qu'il recherche. Dans ce cas là, il va alors chercher les fonctions qui correspondent en utilisant les conversions de type.
Une conversion de type est la transformation d'une variable de type ***Type1*** en la même variable de type ***Type2***.

Il existe de nombreuses conversions de types.
Retenez juste que le compilateur est capable transformer une variable numérique en n'importe quelle variable numérique d'un autre type (même si celle-ci est dégradante).

Ruser pour mieux compiler

Il est alors simple de comprendre comment spécialiser sa fonction pour un type spécial de données.

Prenons le code suivant:

Code : C++

```
#include <iostream>
#include <string>

using namespace std;

template<typename Type> Type calculerSomme(Type operande1, Type
operande2)
{
    Type resultat = operande1 + operande2;
    return resultat;
}
```

```
}

int calculerSomme(string operande1, string operande2)
{
    int resultat;
    resultat = operande1.size() + operande2.size();
    cout << "Utilisation de la spécialisation" << endl;
    return resultat;
}

int main()
{
    int a = 20, b = 10;
    string c = "Bonjour";
    string d = " les zéros!";

    cout << "La somme vaut " << calculerSomme(a, b) << endl;

    cout << "La somme vaut " << calculerSomme(c, d) << endl;

    system("PAUSE");
    return 0;
}
```

Ce code va nous afficher ceci:

Code : Console

```
La somme vaut 30
Utilisation de la specialisation
La somme vaut 18
```

Ici, on veut pouvoir calculer la taille de deux chaînes et en renvoyer la somme.
Mais notre fonction patron n'a pas un algorithme adapté à cette opération.

On écrit donc à nouveau la fonction ***calculerSomme***, adaptée à l'envoi de ***string***. Ainsi lors de `calculerSomme(c, d)`, le compilateur recherche les fonctions qui correspondent exactement à l'appel. C'est ainsi qu'elle trouve notre nouvelle fonction, et pas notre fonction patron

Ici, le compilateur **pourrait** utiliser notre fonction patron.
Cependant il rencontre notre autre fonction ***calculerSomme***, qui correspond exactement à ce qu'il recherche.
Il va donc utiliser cette fonction en priorité.

Pour ne pas faire risquer de faire des erreurs dans la surcharge et l'utilisation de vos fonctions patrons, je vous recommande de vous renseignez sur les conversions de type.
Je ferais peut-être un tutoriel à ce sujet, plus tard.

En attendant, j'espère que ce tutoriel vous a aidé.

Merci de votre *écoute*.

FoeSteR `

Ce tutoriel a été corrigé par les zCorrecteurs.