Fonctions génériques

Intérêt de la programmation générique

La programmation générique consiste à écrire un code qui n'est pas spécifique d'un type particulier, mais peut s'adapter à plusieurs types.

Pour comprendre ce concept, prenez un exemple simple : vous souhaitez écrire une fonction qui réalise une addition. Vous pouvez par exemple écrire :

```
#include <iostream>

int add(int lhs, int rhs) {
    return lhs + rhs;
}

int main() {
    std::cout << add(3, 4) << std::endl;
}
```

affiche :

```
7
```

Rien de compliqué, c'est une fonction classique qui prend deux parametres entiers et retourne un entier.

Si vous ajoutez le second calcul suivant :

```
#include <iostream>

int add(int lhs, int rhs) {
    return lhs + rhs;
}
```

```
int main() {
    std::cout << add(3, 4) << std::endl;
    std::cout << add(1.2, 3.4) << std::endl;
}
```

affiche :

```
main.cpp:9:22: warning: implicit conversion from 'double' to
'int'
changes value from 1.2 to 1 [-Wliteral-conversion]
    std::cout << add(1.2, 3.4) << std::endl;
                 ~~~ ^~~
main.cpp:9:27: warning: implicit conversion from 'double' to
'int'
changes value from 3.4 to 3 [-Wliteral-conversion]
    std::cout << add(1.2, 3.4) << std::endl;
                 ~~~      ^~~
2 warnings generated.
7
4
```

Premièrement, le code émet deux avertissements, du fait de la conversion implicite de `double` en `int`. Et deuxièmement, le résultat obtenu n'est pas correct, la valeur attendue (4.6) est arrondie. La raison est que vous avez écrit une fonction `add` qui utilise des entiers comme parametres. Le compilateur a le choix entre réaliser une conversion des types si c'est possible, ou de produire une erreur si ce n'est pas le cas.

L'appel de la fonction `add` avec des valeurs de type `double` est équivalent au code suivant :

```
const int lhs = 1.2;  // arrondi en 1
const int rhs = 3.4;  // arrondi en 3
add(lhs, rhs);        // calcul 1 + 3
```

Une première solution pour corriger ce problème est d'utiliser une surcharge de fonctions et d'écrire une fonction `add` qui prend un entier en paramètre et une fonction `add` qui prend un réel :

```
int add(int lhs, int rhs) {
    return lhs + rhs;
```

```
}

double add(double lhs, double rhs) {
    return lhs + rhs;
}
```

Dans ce cas, le compilateur n'a pas besoin de faire de conversion, il utilise la fonction correspondante aux types des arguments.

Cependant, cette approche est limitée. Si vous appelez cette fonction avec des arguments de type `short int` ou `float` (par exemple), le résultat sera automatiquement convertie respectivement en `int` et en `double` (par promotion).

Pour éviter cela, il faudra proposer une surcharge de la fonction `add` pour chaque type d'arguments que vous voulez utiliser. Le code n'est pas facilement évolutif, vous devez modifier un code existant si vous ajouter des nouveaux types.

La programmation générique va permettre de résoudre ce problème, en écrivant des fonctions qui prendront plusieurs types de parametres. Un exemple de telles fonctions que vous avez déjà rencontrée est les algorithmes de la bibliothèque standard, qui peuvent être appelés sur plusieurs types de conteneurs.

```
std::string s { "azerty" };
std::sort(std::begin(s), std::end(s));  // ok, trie des
chaines

vector<int> v { 1, 3, 5, 2, 4 };
std::sort(std::begin(v), std::end(v));  // ok, trie des
entiers
```

Définir une fonction template

Dans une fonction template, un ou plusieurs types utilisés dans la fonction (généralement les types des parametres de fonction ou du retour de la fonction) sont remplacés par un parametre template, pouvant representer plusieurs types.

La syntaxe d'une fonction template est la suivante :

```
template<LISTE_PARAMETRES_TEMPLATE>
FONCTION...
```

La première ligne permet de définir un ou plusieurs paramètres template, qui seront utilisés dans la fonction *comme si c'était des types*.

Un paramètre template s'écrit de la façon suivante :

```
typename IDENTIFIANT
```

Le mot-clé `typename` indique que l'identifiant represente un nom de type. L'identifiant respecte les règles habituelles pour écrire un identifiant (contient des lettres minuscules ou majuscules, le caractère `_` ou des chiffre sauf en premiere position).

Une liste de parametres template sera constitué de plusieurs parametres template (mot-clé `typename` et identifiant), séparés par des virgules.

```
typename IDENTIFIANT, typename IDENTIFIANT, typename
IDENTIFIANT (...)
```

Il est classique d'utiliser des majuscules uniquement pour ecrire les parametres template. En particulier, vous verrez souvent des parametres template nommes `T`, `U`, etc. Bien sur, il est préférable de donner des noms les plus expressifs possible.

Par exemple, avec la fonction `add` precedente :

```
template<typename T>
T add(T lhs, T rhs) {
    return lhs + rhs;
}
```

Ce code définit une fonction template qui possède un paramètre template nomme `T`. Ce paramètre template est utilisée trois fois dans la fonction : dans les deux parametres de fonction en entrée et comme type de retour de la fonction.

typename et class

Il est également possible d'utiliser le mot-clé `class` a la place de `typename`. Pour éviter les ambiguïtés avec les classes de la programmation objet, seul le mot-clé `typename`sera utilisée dans ce cours. Mais retenez que vous pouvez rencontrer aussi `class` dans un code pour définir un paramètre template.

Ce code implique que les types des parametres en entrée et en sortie sont le même type : `T` peut être remplacé par n'importe quel type, mais chaque occurrence de `T` correspondra toujours au même type. Par exemple, cette fonction `add` pourra être appelée avec deux entiers et retourner un entier, ou être appelée avec deux réels retourner un réel, mais elle ne pourra pas prendre en paramètre des entiers et retourner des réels.

Il est possible d'utiliser des types différents pour les différents parametres. Par exemple :

```
template<typename T, typename U, typename V>
T add(U lhs, V rhs) {
    return lhs + rhs;
}
```

Ce code utilise trois paramètre template `T`, `U` et `V`, chaque paramètre pouvant être remplacé par un type différent. Par exemple, cette fonction pourra prendre en paramètre un`int` et un `double` et retourner un `float`.

Vous êtes libre de définir autant de paramètre template que vous souhaitez et de les mélanger avec des types concrets.

```
template<typename T, typename U>
T add(double lhs, V rhs) {
    return lhs + rhs;
}
```

Cette fonction a un paramètre template en entrée (qui sera remplacé par n'importe quel type lors de l'appel), un paramètre de type `double` en entrée, et peut retourner n'importe quel type.

Appeler une fonction template

Lors de l'appel d'une fonction template, le compilateur va remplacer les parametres template par des types concret. Cette étape s'appelle l'instanciation des template. A partir d'une fonction template, le compilateur va générer les fonctions concrète correspondant a chaque type concret qui sont utilisés dans les appels de fonction.

Par exemple, pour la fonction `add` precedente, si cette fonction est appelée avec les types `int` et `double`, le compilateur va générer deux fonction surchargées, correspondant a ces types concrets.

```
template<typename T>
T add(T lhs, T rhs) {
    return lhs + rhs;
}

add(1, 2);      // int
add(1.0, 2.0);  // double
```

Le code précédent sera équivalent au code suivant, après l'instanciation des templates par le compilateur :

```
int add(int lhs, int rhs) {           // int remplace T
    return lhs + rhs;
}

double add(double lhs, double rhs) {  // double remplace T
    return lhs + rhs;
}

add(1, 2);      // int
add(1.0, 2.0);  // double
```

Lorsqu'un paramètre template est utilisé dans plusieurs paramètre de fonction (comme c'est le cas avec la fonction `add` precedente), il est nécessaire que les types soient identiques lors de l'appel de fonction, sinon cela produit une erreur.

```
template<typename T>
```

```
T add(T lhs, T rhs) {
    return lhs + rhs;
}

int main() {
    add(1, 1.2);  // int ou double ?
}
```

Affiche l'erreur suivante :

```
main.cpp: In function 'int main()':
main.cpp:7:15: error: no matching function for call to
'add(int, double)'
     add(1, 1.2);  // int ou double ?
               ^
main.cpp:2:3: note: candidate: template<class T> T add(T, T)
 T add(T lhs, T rhs) {
   ^~~
main.cpp:2:3: note:   template argument
deduction/substitution failed:
main.cpp:7:15: note:   deduced conflicting types for
parameter 'T' ('int' and 'double')
     add(1, 1.2);  // int ou double ?
               ^
```

Le compilateur indique qu'il ne trouve pas une fonction `add` pouvant correspondre à l'appel (“no matching function for call”), et qu'il trouve une fonction candidate possible, mais qu'il y a un conflit pour les types (“deduced conflicting types for parameter 'T' ('int' and 'double')”).

Déduction des types et appel explicit

Le code précédent est la façon la plus simple d'appeler une fonction template. La syntaxe est identique à un appel de fonction classique, la seule différence est que le compilateur ajoute deux étapes lors de l'appel :

- la déduction des types : le compilateur regarde les types des arguments

dans l'appel de la fonction et déduit les types à utiliser ; - l'instanciation des template : pour chaque combinaison de types déduits, le compilateur génère une fonction avec des types concrets.

Cependant, il n'est pas toujours possible de déduire les types lors de l'appel de la fonction. La déduction des types n'est possible que pour les parametres template utilisés comme paramètre de fonction, pas les parametres template utilisés en retour de fonction ou dans le corps de la fonction. Vous pouvez egalement souhaitez appeler une fonction template en forçant l'utilisation d'argument template spécifique.

```
template<typename T>
T add(int lhs, int rhs) {
    return lhs + rhs;
}

add(1, 2);  // erreur
```

Dans ce cas, il faut expliciter les types que vous souhaitez utiliser pour l'instanciation des templates. La syntaxe est la suivante :

```
NOM_FONCTION<ARGUMENTS_TEMPLATE>(ARGUMENTS_FUNCTION);
```

La différence avec un appel de fonction classique est donc cette liste d'arguments template ajoutée entre chevrons après le nom de la fonction.

Le code precedent devient, par exemple :

```
template<typename T>
T add(int lhs, int rhs) {
    return lhs + rhs;
}

add<int>(1, 2);     // T == int
add<double>(1, 2);  // T == double
```

Parametres et arguments
Notez la similitude des termes utilisés entre parametres et arguments de fonction et parametres et argument template : les parametres

apparaissent dans la declaration des fonctions et les arguments dans les appels de fonction.

Pour les parametres et arguments de fonction :

```
void f(int a, int b, int x) {}  // a, b et c = parametres de
fonction

int main() {
    f(x, y, z);                 // x, y, z = arguments de
fonction
}
```

Pour les parametres et arguments template :

```
template<typename T, typename U>  // T et U = parametres
template
void f(T a, U b) {}

int main() {
    f<int, double>(x, y);         // int et double =
arguments template
}
```

De la même manière, pour l'exemple précédent qui avait un conflit sur les types :

```
template<typename T>
T add(T lhs, T rhs) {
    return lhs + rhs;
}

int main() {
    add<int>(1, 1.2);  // int !
}
```

Notez que lorsque l'argument template est spécifié, les arguments de fonction sont convertie, si nécessaire, pour s'adapter à l'argument template. Dans ce code, la valeur `1.2` (`double`) est convertie (et arrondie) en `1` (`int`).

Il est possible de mélanger déduction de types et arguments template explicite dans un appel de fonction template. Dans ce cas, les arguments template explicite correspondent aux premiers parametres template, les autres parametres template sont déduits.

```
template<typename T, typename U>
T add(T lhs, U rhs) {
    return lhs + rhs;
}

int main() {
    add(1, 1.2);                // T = int,   U = double
    add<float>(1, 1.2);         // T = float, U = double
    add<float, float>(1, 1.2);  // T = float, U = float
}
```

Lors du premier appel de la fonction `add`, les parametre template `T` et `U` sont déduits des arguments de fonction (`int` et `double`). Lors du deuxième appel, le paramètre template `T` est explicite (`float`), le second paramètre `U` est déduits (`double`). Lors du dernier appel, les deux parametres template sont deduits (`float` et `float`).

Type par defaut

Pour pour les parametres de fonction, il est possible de specifier un paramètre template par défaut lors de la déclaration d'un template. Lorsque l'argument template n'est pas spécifié lors de l'appel de la fonction template et que la déduction des types n'est pas possible, ce type par défaut sera utilisé.

La syntaxe pour indiquer un type par défaut est la suivante :

```
typename PARAMETRE_TEMPLATE = TYPE_DEFAUT
```

Par exemple :

```
template<typename T = int>
T add(int lhs, int rhs) {
```

```
    return lhs + rhs;
}

add<float>(1, 2);  // ok, T = float
add(1, 2);         // ok, T = int
```

A la dernière ligne de ce code, le type de retour n'est pas explicite et ne peut pas être déduit. Le type par défaut (`int`) est donc utilisé.

Surcharge de fonctions

2 etapes : instantiation, puis surcharge

```
#include <iostream>

template<typename T, typename U>
void f(T lhs, U rhs) {
    std::cout << "#1" << std::endl;
}

template<typename T>
void f(T lhs, T rhs) {
    std::cout << "#2" << std::endl;
}

int main() {
    f(1, 1);    // #2
    f(1, 1.2);  // #1
}
```

```
#include <iostream>

template<typename T, typename U>
void f(T lhs, U rhs) {
    std::cout << "#1" << std::endl;
}

template<typename T, typename U = int>
void f(T lhs, T rhs) {
```

```
    std::cout << "#2" << std::endl;
}

void f(int lhs, int rhs) {
    std::cout << "#3" << std::endl;
}

int main() {
    f(1, 2);    // #3
    f(1, 2.0);  // #1
}
```

Echec d'instanciation

Instantie, puis verifie si cela a un sens.

```
template<typename T>
T add(T lhs, T rhs) {
    return lhs + rhs;
}

int main() {
    add("123", "abc");
}
```

```
main.cpp: In instantiation of 'T add(T, T) [with T = const
char*]':
main.cpp:7:21:   required from here
main.cpp:3:16: error: invalid operands of types 'const
char*'
and 'const char*' to binary 'operator+'
     return lhs + rhs;
            ~~~~^~~~
```

SFINAE

Un exemple d'application : les algorithmes de la bibliothèque standard

Idem, avec Itérateur en paramètre template. Prototype :

```
template<typename Iterator>
void sort(Iterator begin, Iterator end);
```

Lorsqu'on appelle cette fonction, le compilateur détermine quel est le type de `Iterator` en fonction des arguments passés :

```
std::vector<int> v {};
std::sort(std::begin(v), std::end(v); // on sait que
Iterator correspond à un itérateur sur un vector<int>
```

Template et meta-programmation

Ce chapitre est une introduction aux fonctions template et à la programmation générique. L'utilisation des template est donc limitée au stricte minimum. Mais il faut savoir que les templates en C++ sont beaucoup plus puissant que cela et forme un véritable langage de programme dans le C++. Ce méta-langage propose les fonctionnalites classiques d'un langage de programmation, en particulier la possibilité de faire des tests et des boucles.

Cette méta-programmation présente un avantage très spécifique : elle ne fonctionne que lors de la compilation, pas lors de l'exécution. Elle permet donc d'écrire du code de haut niveau, qui va adapter le comportement du code en fonction des types, faire des vérifications avancées sur la qualité du code, et cela sans aucun coût à l'exécution du programme.

La méta-programmation est une caractéristique du C++ qui différencie ce langage de la majorité des autres langages de programmation. Son apprentissage n'est pas simple et sera vu dans un autre cours.

Chapitre précédent	Sommaire principal	Chapitre suivant