Les 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 paramètres 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(1.7, 3.8) << std::endl;
}
```

affiche :

```
main.cpp:9:22: warning: implicit conversion from 'double' to
'int'
changes value from 1.7 to 1 [-Wliteral-conversion]
    std::cout << add(1.7, 3.8) << std::endl;
                 ~~~ ^~~
main.cpp:9:27: warning: implicit conversion from 'double' to
'int'
changes value from 3.8 to 3 [-Wliteral-conversion]
    std::cout << add(1.7, 3.8) << 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 (5.5) est arrondie à la valeur 4.

La raison est que vous avez écrit une fonction `add` qui utilise des entiers comme paramètres. 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. Dans le cas present, il choisit la conversion implicite, avec un avertissement.

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

```
const int lhs = 1.7;  // arrondi en 1
const int rhs = 3.8;  // arrondi en 3
add(lhs, rhs);        // calcul 1 + 3
```

Notez que la conversion est réalisée sur les valeurs entrée dans ce cas, ce qui retourne la valeur 4. Si la conversion était réalisée uniquement sur

la valeur de sortie, la valeur 5.5 serait arrondie à la valeur 5.

int add(double lhs, double rhs) { return lhs + rhs; } std::cout << add(1.7, 3.8) << std::endl; // affiche 5

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. Et il devient très vite lourd de devoir écrire toutes les fonctions `add` possibles.

La programmation générique va permettre de résoudre ce problème, en écrivant des fonctions dont les types des paramètres s'adapteront en fonction des arguments utilisés dans l'appel de fonction. 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, tri une
```

```
chaine

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

Définir une fonction template

Dans une fonction template, un ou plusieurs types utilisés dans la fonction (généralement les types des paramètres de fonction ou du retour de la fonction) sont remplacés par un paramètre *template*, pouvant représenter 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 chiffres sauf en premiere position).

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

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

Il est classique d'utiliser des majuscules pour écrire les paramètres template. En particulier, vous verrez souvent des paramètres template nommés `T`, `U`, etc. Bien sur, il est préférable de donner des noms les plus

expressifs possible, mais quand le nom représente “n'importe quoi”, c'est moins problématique.

Par exemple, avec la fonction `add` précédente :

```
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 paramètres 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 paramètres 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 paramètres. 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 paramètres *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 à chaque type concret qui sont utilisés dans les appels de fonction.

Par exemple, si la fonction `add` précédente est appelée avec les types `int` et `double`, le compilateur va générer deux fonction surchargées, correspondant à 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` précédente), il est nécessaire que les types soient identiques lors de l'appel de fonction, sinon la déduction échoué et 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 *templates* : 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 paramètres *template* utilisés comme paramètre de fonction, pas les paramètres *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'arguments *templates* spécifiques.

```
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 pour appeler la fonction 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 les chevrons après le nom de la fonction.

Le code précédent 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
```

Paramètres et arguments
Notez la similitude des termes utilisés entre “paramètre” et “argument” de fonction et “paramètre” et “argument” *template* : les paramètres apparaissent dans la déclaration des fonctions et les arguments dans les appels de fonction.

Pour les paramètres et arguments de fonction :

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

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

Pour les paramètres et arguments *template* :

```
template<typename T, typename U>  // T et U = paramètres
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 (il ne peut pas y avoir d'échec de déduction des types, puisque cette étape n'est pas réalisée). 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 paramètres template, les autres paramètres 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 paramètres template sont déduits (`float` et `float`).

Type par defaut

Pour pour les paramètres de fonction, il est possible de spécifier 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

Les fonctions template peuvent être surchargées, entre elles et avec les fonctions classiques. Cette fois-ci aussi, le compilateur travaille par étapes :

- le compilateur instancie les fonctions *templates* ;
- puis la résolution de la surcharge est réalisée sur l'ensemble des fonctions.

La résolution de la surcharge est similaire à celle sans fonction template, les fonctions template ont une priorité intermédiaire entre les fonctions sans conversion et avec conversion :

- appel de fonction sans aucune conversion ;
- appel de fonction template
- appel de fonction avec promotion ;
- appel de fonction avec conversion.

Voici un exemple pour que les choses soient plus concrètes.

```
#include <iostream>

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

void f(int lhs, int rhs) {
    std::cout << "#2" << std::endl;
}

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

Dans ce code, le premier appel de la fonction `f` contient deux arguments de type `int`. La fonction fonction template `#1` sera instancié avec le paramètre `T = int`, ce qui produira une fonction avec la signature suivante : `f(int, int)`.

Lors de la résolution de la surcharge, les deux fonctions ont donc la même signature, mais l'une est une instance de fonction *template* et n'est donc pas prioritaire. La fonction `#2` est donc choisie par le compilateur.

Dans le second appel de la fonction `f`, celle-ci contient des arguments de types `int` et `double`. La fonction *template* sera donc instanciée avec la signature suivante : `f(int, double)`. Lors de la résolution de la surcharge, le compilateur a le choix entre une fonction *template* dont les types des paramètres correspondent aux arguments, et une fonction non template qui nécessite une conversion de `double` en `int` pour être appelée. La fonction template `#1` est donc choisie.

Echec d'instanciation

Lors de l'utilisation des *templates*, vous pouvez rencontrer deux types d'erreurs :

- lors de la déduction des types, comme vous l'avez précédemment (“template argument déduction/substitution failed”) ;
- lors de l'appel de la fonction après instanciation.

En effet, comme le compilateur fonctionne par étape, il ne se préoccupe pas du corps de la fonction lorsqu'il instancie les fonctions templates. Il est donc possible qu'une fonction template soit instanciée, mais que la fonction n'a pas de sens.

Par exemple :

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

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

Dans ce code, les arguments de la fonction `add` sont des littérales chaînes (`const char*`) et la fonction template est instanciée avec la signature suivante : `add(const char*, const char*)`. Cette étape ne pose pas de problème de déduction des types et aucune erreur s'est

produite.

Le code est donc equivalent a :

```
const char* add(const char* lhs, const char* rhs) {
    return lhs + rhs;
}

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

Et la se pose un problème : l'opérateur d'addition `+` n'a pas de sens pour le type `const char*`. Ce code produit donc l'erreur suivante (“invalid operands ... to binary operator+”) :

```
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;
            ~~~~^~~~
```

Assertion sur les types

Vous connaissez déjà les assertions, qui permettent de vérifier à l'exécution qu'une condition est vraie.

```
#include <cassert>

int main() {
    assert(1 == 1);  // ok
    assert(1 == 2);  // produit une erreur
}
```

Affiche :

```
a.out: main.cpp:5: int main(): Assertion `1 == 2' failed.
```

Il existe un second type d'assertion, qui permet de vérifier une condition à la compilation. Comme les types (et donc les *templates*) sont résolues à la compilation, ce type d'assertion va permettre d'imposer des conditions sur les parametres templates.

La syntaxe est la suivante :

```
static_assert(CONDITION, MESSAGE);
```

Note : contrairement à `assert` qui nécessaire d'inclure un fichier d'en-tête (`<cassert>`), `static_assert` est un mot-clé du langage et ne nécessite pas d'inclusion.

Et pour écrire des conditions sur les types, vous pouvez utiliser les fonctionnalités permettant d'obtenir des informations sur les types que vous avez vu dans le chapitre : Obtenir des informations sur les types.

Pour prendre un exemple concret, imaginez que vous souhaitez limiter l'utilisation de votre fonction `add` uniquement aux types représentant un nombre (et donc interdire l'utilisation de votre fonction avec une chaine de caracteres par exemple). Vous pouvez pour cela utiliser `static_assert` avec `is_arithmetic`.

main.cpp
```cpp
#include <type_traits>

template<typename T>
T add(T lhs, T rhs) {
    static_assert(std::is_arithmetic<T>::value, "T is not
arithmetic type!");
    return lhs + rhs;
}

int main() {
    add(1, 2);          // ok
    add("123", "abc");  // erreur
}
```

Dans le premier appel à la fonction `add`, le paramètre *template* `T` est instancié en utilisant le type `int`. L'assertion est vraie et cela ne produit pas d'erreur.

Le second appel n'est pas un type de nombre et produit une assertion :

```
main.cpp:5:5: error: static_assert failed "T is not
arithmetic type!"
    static_assert(std::is_arithmetic<T>::value, "T is not
arithmetic type!");
    ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.cpp:11:5: note: in instantiation of function template
specialization 'add<const char *>' requested here
    add("123", "abc");  // erreur
    ^
```

Une assertion statique accepte n'importe quelle expression qui retourne un booléen a la compilation, vous pouvez en particulier utiliser les opérateurs logiques `!` (NON), `&&` (ET) et `||` (OU)

Substitution failure is not an error (SFINAE)

Derrière cet acronyme un peu étrange se cache en fait un concept assez simple. Prenons le code suivant, que vous avez déjà vu :

```
#include <iostream>

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

void f(int lhs, int rhs) {
    std::cout << "#2" << std::endl;
}

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

Ce code ne produit pas d'erreur.

Le même code, sans la seconde fonction `f` :

```
#include <iostream>

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

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

Ce code produit une erreur, puisque les arguments de la fonction sont de types differents (`int` et `double`).

```
main.cpp:9:5: error: no matching function for call to 'f'
    f(1, 2.0);  // #2
    ^
main.cpp:4:6: note: candidate template ignored: deduced
conflicting
types for parameter 'T' ('int' vs. 'double')
void f(T lhs, T rhs) {
     ^
```

Vous pouvez alors vous demander pourquoi la fonction *template*, qui est manifestement problématique, ne produit également pas une erreur dans le premier code ?

La raison est que l'échec de l'instanciation des template ne produit pas d'erreur (c'est ce que signifie “Substitution failure is not an error” : “l'echec d'une substitution n'est pas une erreur”). Ce qui produit une erreur est le fait que le compilateur ne trouve aucune fonction valide dans le second code, pas l'échec de l'instanciation.

Dans cet exemple, le SFINAE a ete utilise sans le savoir. Mais il existe de nombreuses techniques de méta-programmation qui utilise ce concept. Cela signifie en particulier que vous pouvez écrire autant de fonctions

template que vous voulez, du moment qu'au moins une fonction est valide, le code compilera sans erreur.

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

Supposez que vous écrivez un algorithme qui prend en paramètre une collection de bibliothèque standard (par exemple `std::vector<int>`). Vous pourriez écrire une fonction qui prend en parametre cette collection. Par exemple :

```
void do_something(const std::vector<int>& v);
```

Comme ce chapitre est consacré à la généricité, vous avez surement compris le problème : ce code n'est pas générique (vous ne pouvez pas l'utiliser avec n'importe quel type de collection).

Vous pouvez améliorer les choses en transformant cette fonction en template. Par exemple :

```
template<typename T>
void do_something(const std::vector<T>& v);
```

Cette fonction est un peu plus générique, il est maintenant possible de changer le type d'éléments dans la collection. Cependant, ce n'est pas encore totalement générique : il n'est pas possible de changer le type de collection, cela sera forcement un `std::vector`.

Vous pourriez alors écrire le code suivant :

```
template<typename T>
void do_something(const T& v);
```

Ce code est totalement generique... et peut être un peu trop : la notion de collection est perdue, `T` peut représenter n'importe quel type autre qu'une collection.

Pour résoudre cette problématique, la bibliothèque standard implémente

le concept d'itérateur (que vous avez déjà vu). Pour rappel, un itérateur est une indirection sur un élément d'une collection.

Vous avez vu dans le chapitre sur les algorithmes de la bibliothèque standard que ceux-ci peuvent s'adapter a differents types de collections de donnees.

```
std::vector<int> v_int { 1, 2, 3 };
std::sort(std::begin(v_int), std::end(v_int);  // ok

std::vector<std::string> v_str { "abc", "123" };
std::sort(std::begin(v_str), std::end(v_str);  // ok
```

Comme vous pouvez vous en douter, cela est possible en utilisant les *templates*.

La signature de la fonction `std::sort` est une fonction *template*, qui prend en parametre *template* le type d'iterateurs (voir la documentation sur cppreference : sort) :

```
template<typename RandomIt>
void sort(RandomIt first, RandomIt last);
```

Dans ce code, le paramètre *template* a été nommé `RandomIt`, pour indiquer que c'est un itérateur de type “random” (pour rappel, voir le chapitre Les catégories d'itérateurs).

Lorsque vous écrirez vos propres algorithmes, essayez de respecter cette signature pour les fonctions. Cela permettra la plus grande flexibilité dans le code et garantira que vos algorithmes soient compatibles avec les collections de la bibliotheque standard.

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