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 pourra s'adapter à plusieurs types.

Pour comprendre ce concept, prenez un exemple simple : vous souhaitez écrire une fonction qui calcule 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 affiché deux avertissements, du fait de la conversion implicite de `double` en `int`, ce qui produit un arrondi des valeurs. Et deuxièmement, cela retourne une valeur entière arrondie et pas le resultat attendu (la valeur “4.6”).

La raison est qu'il n'existe pas de fonction `add` qui prend en arguments des types réels. Le compilateur ne trouve que la fonction `add` pour des entiers. Il regarde donc s'il peut faire une conversion, ce qui est le cas. Le code de l'appel de la fonction `add` avec les arguments réels est donc 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 :

```
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 appellee sur plusieurs types de conteneurs.

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

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

Les fonctions template

`auto` n'est pas pris en charge par tous les compilateurs. Possibilité d'expliciter le type générique en utilisant des fonctions template.

Une fonction classique permet de passer des données en paramètre. Les fonctions template vont plus loin, elles permettent de passer des types comme paramètre. C'est-à-dire que les types manipulés par un template ne sont pas fixés (`int`, `double`, etc), mais sont des paramètres.

Vous avez déjà vu des template dans ce cours, le meilleur exemple est `std::vector` et `std::array`. Ces classes template représentent des collections pouvant contenir n'importe quel type de données. Le type manipulé dans la collection est indiqué dans les chevrons :

```
std::vector<int> ints {};    // tableau de int
std::vector<double> doubles {}; // tableau de double
```

Pour définir une fonction template, la syntaxe :

```
template<paramètres template>
paramètre_retour nom_fonction(paramètres de fonction) {
}
```

On voit ici qu'une fonction template prend deux types de paramètre :

- les paramètres template, qui sont des types et sont évalués à la compilation. Mot-clé “typename” ou “class” suivi d'un nom de paramètre ;
- les paramètres de fonction, qui sont des valeurs et sont évaluées à l'exécution (sauf `constexpr`...)

Les paramètres template déclarés entre les chevrons peuvent ensuite être utilisés dans la fonction (même dans les paramètres de fonction et le type de retour de fonction).

Par exemple, pour la fonction `add`, on peut écrire :

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

On déclare ici un paramètre template qui se nomme “T”, que l'on utilise comme retour de fonction et comme type pour les paramètres de

fonction (n'oubliez pas que `T` représente un type, pas une variable).

Pour appeler une fonction template, en spécifiant les arguments :

```
nom_fonction<arguments template>(arguments de fonction);
```

Les arguments template sont les types qui seront utilisés pour appeler la fonction. Par exemple, écrire `add<int>` permet au compilateur de remplacer `T` par `int` dans le code précédent, qui devient :

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

De même si on écrit `add<double>` ou n'importe quoi d'autre.

On peut donc appeler cette fonction `add` avec différents types d'arguments :

```
int i = add<int>(1, 2);
double d = add<double>(1.2, 3.4);
```

Le type `T` est défini uniquement dans la fonction template, donc il n'est pas possible d'écrire :

```
T i = add<int>(1, 2);
T d = add<double>(1.2, 3.4);
```

(cela n'aurait pas de sens, le compilateur ne sait pas si `T` = `int` ou `double`). Il est possible d'utiliser l'inférence de type :

```
auto i = add<int>(1, 2);
auto d = add<double>(1.2, 3.4);
```

Dans ce cas, le compilateur sait déterminer le type.

déduction automatique des arguments template

différence avec `auto` → un seul type pour `lhs` et `rhs`, `add(1, 1.2)` pose problème. Possible écrire :

```
template<typename T1, typename T2, typename T3>
T1 add(T2 lhs, T3 rhs);
```

Nécessite de mettre `T1` au moins (ne peut pas être déduit)

Possible aussi de ne pas mettre `common_type` :

```
template<typename T1, typename T2>
std::common_type<T1, T2> add(T1 lhs, T2 rhs);
```

plus besoin de spécifier le type de retour

note : template != generique. Plus puissant, langage complet (langage dans un langage, turing complet, meta programmation).

Les algorithmes de la bibliothèque standard

Idem, avec Itérateur en paramètre template. Prototype :

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

Lorsque l'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>
```

Chapitre précédent	Sommaire principal	Chapitre suivant