

# Fonctions génériques

## Problématiques

On souhaite écrire une fonction qui fait une addition. On peut écrire par exemple :

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

Ce code fonctionne et donne le résultat attendu :

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

affiche :

7

Maintenant, si on écrit :

```
int main() {  
    cout << add(1.2, 3.4) << endl;  
}
```

Ce code affiche :

4

au lieu de "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, et donc le fait. Code équivalent à :

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

Une première solution est d'utiliser une surcharge et d'écrire deux 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 trouve les 2 fonctions, la première a besoin d'une conversion, l'autre non. Il choisit donc celle qui ne nécessite pas de conversion.

On comprend vite le problème de cette approche. Si on veut utiliser cette fonction avec 10 types différents, il faudra écrire 10 fonctions surchargées différentes.

Si on met ce code dans une bibliothèque et que l'on veut l'utiliser sur un 11<sup>e</sup> type que l'on avait pas prévu, il faudra modifier le code existant, c'est à dire modifier la bibliothèque que l'on n'a pas écrite.

Ce code n'est donc pas générique, il est compliqué de le faire évoluer.

Si on regarde les algorithmes de la bibliothèque standard, on voit qu'ils sont conçus pour pouvoir être utilisés sur n'importe quel type de conteneur. Il est donc possible d'écrire un code générique, qui s'adapte au type que l'on utilise.

## Les fonctions template

`auto` n'est pas pris en charge par tous les compilateurs. Possibilité d'explicitier 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 évalués 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

## 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](#)