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 surchage 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 mets ce code dans une bibliothèque et que l'on veut l'utiliser sur un 11ème 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 écrit.

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

Si on regarde les algorithmes de la bibliothèque standard, il voit qu'ils sont conçu pour pouvoir être utilisé 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.

L'inférence de type

En fait, vous avez déjà vu une solution possible dans les chapitres précédents. Lors que l'on a vu les fonctions lambdas, vous avez utiliser l'inférence de type et le mot clé `auto` pour laisser le compilateur choisir

les types des paramètres de fonction.

Cette approche est également utilisable avec les fonctions : uniquement gcc 4.9 pour le moment, cf TS concept

```
#include <iostream>

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

int main() {
    std::cout << add(1, 2) << std::endl;
    std::cout << add(1.2, 3.4) << std::endl;
}
```

Dans le premier cas, le compilateur déduit que l'on souhaite utiliser des entier et détermine que `auto` est équivalent à `int`. Dans le second cas, il utlise `double`.

De la même manière, il détermine que le type de retour de la fonction est de même type que l'expression `lhs + rhs`. Comme ces deux variables sont de même type dans le code d'exemple, le compilateur détermine sans problème que le résultat doit être de même type.

Que ce passe-t-il lorsque les deux types ne sont pas identique. Par exemple, si on force l'un des deux paramètres :

```
#include <iostream>

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

int main() {
    std::cout << add(1, 2) << std::endl;
    std::cout << add(2, 3.4) << std::endl;
}
```

affiche :

```
3
5.4
```

Le compilateur réussit à déterminer le type correcte de l'expression. Lorsque l'on utilise deux entiers, le résultat est un entier et lorsque l'on utilise un entier et un réel, le résultat est un réel.

std::common_type ??

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 paramètres. Les fonctions template vont plus loin, elles permettent de passer des types comme paramètres. C'est à dire que les types manipulés par un template n'est pas fixé (int, double, etc), mais est un paramètre.

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 :

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

Pour définir une fonction template, la sytnaxe :

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

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

- les paramètres template, qui sont des types et sont évaluer à la compilation. mot clé “typename” ou “class” suivi d'un nom de paramètre ;

- les paramètres de fonction, qui sont de valeurs et sont évaleur à l'exécution (sauf constexpr...)

Les paramètres template déclarés entre les chevrons peuvent ensuite être utilisé 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'arguemnts :

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

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

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

Dans ce cas, le compilateur sait déterminer le type.

déduction automatique des arugment tempalte

différence avec auto → un seul type pour lgs et rgs, 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>
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 Iterateur en paramètre template. Prototype :

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

Lorsque l'on appelle cette fonction, le compilateur détermine que est le type de Iterator, en fonction des arguments passé :

```
vector<int> v {};
```

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


Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++