Les foncteurs

foncteur ? fonction objet ? opérateur ? autre ?

Les foncteurs par défaut

Vous avez vu dans le chapitre précédent comment utiliser la fonction `std::sort` pour trier une collection :

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

std::string s { "hello, world!" };
std::sort(begin(s), end(s));
```

Cet algorithme est dit “modifiant” puisqu'il modifie directement le conteneur sur lequel on applique la fonction.

Fondamentalement, cet algorithme fonctionne de la façon suivante : il parcourir les éléments de la collection et réalise des tests de comparaison par paire d'éléments. Pour faire cette comparaison, l'algorithme utilise l'opérateur de comparaison `<` sur les éléments. Par exemple, pour faire le trie d'un tableau d'entiers (`vector<int>`), l'algorithme réaliser des comparaisons d'entiers (`valeur 1 < valeur 2`).

Dit autrement, cela veut dire que si on utilise un `vector<un_type>`, il faut que la comparaison `<` est un sens pour ce type `un-type` (ce qui sera le cas avec la majorité des types de base du C++).

On dit que l'opérateur `<` est le prédicat utilisé par l'algorithme de trie `std::sort`. Plus généralement, un prédicat est une expression qui retourne un booléen (`true` ou `false`). Les différents algorithmes de la bibliothèque standard n'utilisent pas tous l'opérateur `<`, certains utilisent l'opérateur d'égalité `==`, d'autres n'utilisent pas de prédicat.

Imaginons maintenant que l'on souhaite trier une collection dans l'ordre inverse, c'est-à-dire du plus grand au plus petit. Une première solution serait de trier dans l'ordre par défaut (plus petit au plus grand), puis d'inverser l'ordre des éléments. Une autre solution serait de réécrire un algorithme de trie (appelé `reverse_sort` par exemple) et qui trie dans l'ordre inverse (du plus grand au plus petit).

Ces deux solutions ne sont pas correctes en termes de C++ moderne. La première est inutilement plus compliqué (il faut écrire deux lignes au lieu d'une seule), la seconde demande de réécrire l'algorithme de trie.

Les foncteurs de la bibliothèque standard

Heureusement, la bibliothèque standard a été conçue pour être le plus générique possible, suivant les principes de la programmation moderne. Pour cela, la majorité des algorithmes de la bibliothèque standard existe en deux versions. La première utilise les foncteurs par défaut, la seconde admet un argument supplémentaire permettant de fournir un foncteur personnalisé. Par exemple, la fonction `sort` peut s'utiliser avec le prédicat par défaut (utilisation de `<`) ou un foncteur personnalisé :

```
std::sort(begin(v), end(v));              // foncteur par défaut
std::sort(begin(v), end(v), un_foncteur); // foncteur personnalisé
```

Les cas les plus génériques, comme trier du plus grand au plus petit, sont déjà implémentés dans la bibliothèque standard. Ces prédicats sont définis dans le fichier d'en-tête `<functional>`. Par exemple, pour trier du plus grand au plus petit, il est possible d'utiliser le prédicat `greater` (“plus grand que”) de la façon suivante :

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
```

```
int main() {
    std::vector<int> v { 1, 5, 2, 4, 3 };
    std::sort(begin(v), end(v));
    for (auto const value: v)
        std::cout << value << std::endl;
    std::cout << std::endl;

    std::sort(begin(v), end(v), std::greater<int>());
    for (auto const value: v)
        std::cout << value << std::endl;
}
```

affiche :

```
1
2
3
4
5

5
4
3
2
1
```

Le prédicat `greater` est une classe template, il faut préciser le type que l'on souhaite comparer comme argument template (donc entre chevrons). Les parenthèses permettent d'instancier un objet à partir de la classe `greater`. Vous n'avez pas encore vu les classes, vous verrez cela en détail dans la partie sur la programmation objet. Pour le moment, le plus important est de se souvenir de la syntaxe.

Vous pouvez consulter la liste des différents foncteurs de la bibliothèque standard dans la page de documentation Function objects.

Les prédicats sont les suivants :

- Comparaisons
  - `equal_to` pour tester l'égalité `==` ;
  - `not_equal_to` pour tester la différence `!=` ;

    - `greater` pour la comparaison `>` ;
    - `less` pour tester l'égalité `<` ;
    - `greater_equal` pour tester l'égalité `>=` ;
    - `less_equal` pour tester l'égalité `⇐`.
- Opérations logiques
    - `logical_and` pour AND &&
    - `logical_or` OR ||
    - `logical_not` Not !x

bind ?

Les fonctions lambdas

Si vous avez regardé la documentation des foncteurs de la bibliothèque standard, vous avez peut être compris que ce sont de simples classes. Vous apprendrez dans la partie sur la programmation orientée objet comment créer vos propres foncteurs (que l'on appelle aussi “function object”). Mais il est possible de créer des foncteurs plus simplement, en utilisant des fonctions lambdas.

Il est important que vous sachiez créer des fonctions, c'est un point fondamental en C++, vous les utiliserez dans tous vos codes. Et plus important, ce qui sera fondamental est de savoir découper correctement les problèmes complexes en fonctions plus simples. Ce chapitre ne sera pas suffisant pour étudier toutes les possibilités offertes par les fonctions, nous reviendrons dessus en détail par la suite. Cette partie se focalise sur l'utilisation simple des fonctions lambdas avec les algorithmes de la bibliothèque standard.

Les fonctions lambdas sont une technique issue de la programmation fonctionnelle. Vous avez déjà utiliser des fonctions (membres ou libres) et vous avez déjà définit une fonction : la fonction `main`. Pour rappel, une fonction permet de réaliser une tâche particulière et est constituée de :

- un nom de fonction (`main`) ;
- des valeurs optionnelles (arguments) en entrée et sortie ;
- un corps de la fonction, entre crochets, qui contient les

instructions à exécuter lorsque l'on appelle la fonction ;
- il n'est pas possible de définir une fonction dans une autre fonction.

La fonction `main` est une fonction particulière :

- elle est obligatoire ;
- elle ne peut être appelée que par le système (jamais par l'utilisateur) ;
- elle doit être unique ;
- sa signature (c'est-à-dire sa façon d'être écrite) est définie par la norme C++.

```
int main() {
    // corps de la fonction main
}
```

`Programmation fonctionnelle`
Une fonction lambda est une fonction particulière. Elle n'a pas de nom et peut être déclarée dans le corps d'une autre fonction. A part cela, elle se comporte comme une fonction classique et peut recevoir des arguments, retourner une valeur. La définition d'une fonction lambda se décompose en trois parties :

```
[capture](paramètres){corps de la fonction}
```

Les paramètres de fonction

La partie “capture” et “paramètres” permettent toutes les deux de passer des informations dans la fonction. Vous verrez par la suite la différence entre ces deux parties, pour le moment, nous allons nous concentrer sur les paramètres, sans utiliser de capture.

Les paramètres de fonctions sont donc des informations que l'on passe dans une fonction. Ces paramètres s'écrivent comme des variables (un

type et un nom), séparés par des virgules.

coincé, beaucoup de chose à dire sur les fonctions...

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++