[Aller plus loin] Les fonctions comme données

En première approche, une fonction peut être vu comme une boîte noire, qui peut prendre des informations (les arguments), réaliser une tâche et peut retourner un résultat. Les informations en entrée et sortie seront généralement des données plus ou moins complexes : des entiers, des réels, des chaînes, etc.

En fait, les fonctions elle-même sont des informations. Elles peuvent être manipuler comme n'importe quelle donnée, être conservées dans des variables ou être utilisées comme arguments de fonctions. Cela permet de modifier le comportement du programme par le programme lui-même.

Un exemple dans la bibliothèque standard

Comme vous l'avez vu, la syntaxe générale pour appeler une fonction est la suivante :

```
NOM_FONCTION(ARGUMENTS...)
```

En fait, les fonctions ne sont pas les seules syntaxes en C++ qui peuvent s'appeler de cette façon. L'ensemble de syntaxes qui peuvent être appelées de cette façon sont appelés des *callables*, ce qui peut être traduit par “appelables”. Ce terme n'est pas vraiment correct en français, le terme anglais est probablement préférable.

Vous avez déjà rencontré des *callables* qui ne sont pas des fonctions : les foncteurs (aussi appelé fonction-objet) de la bibliothèque standard, comme `std::less` (“plus petit que”) ou `std::plus`.

```
std::less<int>(12, 34);       // est equivalent a (12 < 34)
std::plus<float>(1.2, 3.4);   // est equivalent a (1.2 + 3.4)
```

Dans ce code, `std::less` et `std::plus` sont des classes (des structures de données) et non des fonctions. Mais il est possible de les appeler comme des fonctions.

Les algorithmes de la bibliothèque standard sont des bons exemple d'algorithmes dont le comportement peut être modifié en utilisant des fonctions comme arguments.

Prenez par exemple l'algorithme `std::sort`. Cet algorithme parcourir une collection et compare les elements deux par deux. Par défaut, il utilise le prédicat `std::less`, qui retourne vrai si le premier argument est plus petit que le second argument. (Pour rappel, un prédicat est un foncteur qui retourne un booléen). Le résultat est une collection dont les éléments sont triés du plus petit au plus grand.

```
std::sort(std::begin(v), std::end(v));
```

Pour trier une collection du plus au plus petit, une solution serait d'écrire un nouvel algorithme, qui utilise `std::greater` (“plus grand que”) comme prédicat. Mais cette solution n'est pas très évolutive : il faut écrire un nouvel algorithme a chaque fois que vous avez besoin d'un nouveau prédicat.

Une meilleure solution est de faire en sorte que le prédicat soit un argument de l'algorithme. Par exemple, pour trier du plus grand au plus petit, il est possible d'utiliser `std::greater` avec `std::sort`.

```
std::sort(std::begin(v), std::end(v), std::greater);
```

Il est possible d'aller plus loin et de créer de nouveaux prédicats, qui seront utilisables directement dans les algorithmes standards. Par exemple, si vous créez une structure de données :

```
struct Personne {
    std::string name { "" };
    int age { 0 };
};

bool less_by_name(Person const& lhs, Person const& rhs) {
    return (lhs.name < rhs.name);
```

```
}

bool less_by_age(Person const& lhs, Person const& rhs) {
    return (lhs.age < rhs.age);
}

std::vector<Person> persons = make_persons();
std::sort(std::begin(persons), std::end(persons),
less_by_name); // tri selon le nom
std::sort(std::begin(persons), std::end(persons),
less_by_age);  // tri selon l'age
```

Cette approche permet d'écrire du code fortement réutilisable :

- vous écrivez des structures de données indépendamment des algorithmes ;
- vous écrivez des algorithmes indépendamment des structures de données ;
- vous écrivez des prédicats qui font le lien entre algorithmes et structures de données.

Créer une variable pour une fonction

La syntaxe pour créer une variable (ou un paramètre de fonction) contenant une fonction est strictement identique aux autres types de variables :

```
TYPE NOM_VARIABLE { VALEUR };
```

La différence avec une variable classique est ce que vous allez utiliser pour `TYPE` et `VALEUR`.

Deduction de types

La méthode la plus simple, comme souvent, est de laisser le compilateur faire le travail. Il est tout à fait possible de créer une variable contenant

une fonction, en utilisant la déduction de type `auto` ou `decltype`.

main.cpp

```
#include <iostream>
#include <functional>

template<typename T, typename F>
void apply(T x, T y, F f) {
    std::cout << f(x, y) << std::endl;
}

void f() {
    std::cout << "f()" << std::endl;
}

int minus(int lhs, int rhs) {
    return lhs - rhs;
}

int main() {
    auto x = f;
    x();

    auto y = [](){ std::cout << "lambda" << std::endl; };
    y();

    apply(1, 3, std::plus<int>());
    apply(1, 3, minus);
    apply(1, 3, [](int lhs, int rhs){ return lhs * rhs; });
}
```

La variable `x` est initialisée avec la fonction `f`.

main.cpp

```
#include <iostream>

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

int main() {
```

```
    auto f { add };
    auto x = f(12, 34);
    std::cout << x << std::endl;
}
```

Type explicite de fonction

exemple : bool f(int, double)

En C++, std::function. Template, utiliser signature de fonction. std::function<bool (int, double)>

Note : pointeur de fonction : un peu complexe, heritage du C. bool (*)(int, double);

autres

std::bind = cree nouvelle fonction, avec arguments

std::invoke = appel une fonction avec argument

callback

Chapitre précédent	Sommaire principal	Chapitre suivant