[Aller plus loin] Manipuler une fonction dans une variable

En première approche, une fonction peut être vue 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 elles-mêmes sont des informations. Elles peuvent être manipulées 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 concret d'une telle utilisation. Imaginez que vous travaillez sur un jeu vidéo et vous souhaitez donner la possibilité au joueur de pouvoir changer les actions selon les touches du clavier. Par exemple que la touche A puisse servir à sauter, à avancer ou à tirer, selon le choix du joueur. Une solution est de créer des fonctions `sauter`, `avancer` et `tirer` et d'avoir une variable `touche_A`. Il suffit ensuite de sélectionner la fonction à affecter à la variable `touche_A` et à chaque fois que le joueur appuie sur cette touche, le jeu exécute l'action correspondante.

Vous pourrez aussi voir le terme *callback* (fonction de rappel) pour désigner cette approche.

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 des syntaxes qui peuvent être appelées de cette façon est appelé *callable*, ce qui peut être traduit par “appelable”. 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 équivalent à (12 < 34)
std::plus<float>{}(1.2, 3.4);  // est équivalent à (1.2 +
3.4)
```

Dans ce code, `std::less` et `std::plus` sont des classes template (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 exemples d'algorithmes dont le comportement peut être modifié en utilisant des fonctions comme arguments.

Prenez par exemple l'algorithme `std::sort`. Cet algorithme parcourt une collection et compare les éléments deux par deux. Par défaut, il utilise le prédicat `std::less`, qui retourne `true` si le premier argument est plus petit que le second. (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 grand 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 à 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<type>{});
```

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'âge
```

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`.

Déduction 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`. Dans ce cas, la valeur sera l'identifiant de la fonction (son nom, donc sans les parenthèses, les paramètres de fonction ou le type de retour).

```
void f() {}

auto g = f;
```

Dans ce code, `g` est une variable qui contient la fonction `f`. Utiliser `g` revient donc à utiliser `f`. `g` n'est pas une fonction, c'est une variable. Mais comme elle “contient” une fonction, elle devient un objet appelable et vous pouvez utiliser des parenthèses pour l'appeler. (Une variable ne contient pas à proprement parlé une fonction, mais les détails techniques n'ont pas d'intérêt pour le moment).

main.cpp

```
#include <iostream>

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

int main() {
    auto g = f;
    g();
```

```
}
```

affiche :

```
f()
```

Dans le cas d'une fonction qui possède des paramètres et un retour, l'appel de la variable est identique à un appel direct de la fonction.

main.cpp
```
int f(int i) {
    std::cout << "f:" << i << std::endl;
    return 123;
}

auto g = f;
std::cout << "g:" << g(-1) << std::endl;
```

affiche :

```
f:-1
g:123
```

Cette approche est particulièrement intéressante avec une fonction lambda.

```
auto f = [](int i){ std::cout << i << std::endl; };
```

Les fonctions génériques

Dans le cas ou vous souhaitez utiliser une fonction comme argument d'une autre fonction, vous ne pouvez pas utiliser `auto` en paramètre (cela viendra dans une prochaine norme du C++). Il faut donc recourir aux fonctions génériques et aux *templates*.

La syntaxe d'une fonction générique qui prend une fonction en paramètre est strictement identique aux fonctions génériques classiques : un paramètre template peut représenter aussi bien un type de données

qu'une fonction. (Pour être plus précis, un paramètre template peut représenter n'importe quel type. Et une fonction est un type comme les autres).

```
template<typename F>
void g(F f) {
    f(2);
}
```

La fonction `g` prend en paramètre un type qui devra être appelable (puisque que `f` est utilisé comme une fonction dans le corps de `g`) et qui peut prendre en argument la valeur `2` (donc avoir paramètre de type entier ou convertible depuis un entier. Elle peut également avoir d'autres paramètres, mais ils doivent avoir des paramètres par défaut).

Note : il est classique de nommer un paramètre template `F`, `G`, etc. pour les distinguer des autres types `T`, `U`, etc. C'est simplement une convention d'écriture, le compilateur ne vérifie pas cela.

Par exemple :

```
void print(int i) {
    std::cout << i << std::endl;
}

void signed(int i) {
    std::cout << (i > 0 ? "positive" : "negative") << std::
endl;
}

g(print); // affiche "2"
g(assert) // affiche "positive"
```

Voici un exemple plus intéressant. La fonction `invoke` (qui sera ajoutée dans le C++17) prend en paramètre un objet appelable et des valeurs et appelle cet objet en utilisant les valeurs. La fonction `std::invoke` du C++17 peut s'utiliser avec une liste quelconque de valeurs, mais cet exemple se limitera à une liste déterminée de valeurs (il faut utiliser des *template variadic* pour faire cela).

La fonction `invoke` peut s'écrire de la façon suivante :

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

Note : il est classique aussi de mettre le paramètre template correspondant à un objet appelable en premier.

Pour appeler cette fonction, il faut donner un objet appelable en argument. Cet objet peut être une fonction, un foncteur, une fonction lambda.

main.cpp

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

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

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

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

Type explicite de fonction

Dans certain cas, vous ne pourrez utiliser ni la déduction de type, ni une fonction template (par exemple dans les classes). Il faut dans ce cas écrire explicitement le type de la fonction pour créer une variable. Historiquement, le type d'une fonction vient du langage C et correspond

à un pointeur de fonction (une adresse en mémoire de l'ordinateur). La syntaxe est complexe et l'utilisation de pointeurs pose souvent des problèmes.

Le C++11 a heureusement ajouté des fonctionnalités dans la bibliothèque standard pour simplifier la création de fonctions : `std::function`. Cette classe est définie dans le fichier d'en-tête `<functional>`. C'est une classe *template*, donc il va falloir utiliser les chevrons encore une fois.

La syntaxe générale est la suivante :

```
std::function< TYPE_RETOUR ( LISTE_TYPES ) >
```

Il faut commencer par donner le type de retour `TYPE_RETOUR` ou `void` si la fonction ne retourne rien, puis la liste des types des paramètres `LISTE_TYPES` séparées par des virgules (ou rien si la fonction ne prend aucun paramètre).

Par exemple, pour une fonction qui prend en paramètre un entier, un double et retourne une chaine :

```
std::function< std::string ( int,              double) >
               ^ retour
                           ^ 1er paramètre
                                               ^ 2nd
paramètre
```

Voici quelques exemples :

`std::function`	**Exemple de fonction**
`std::function<void()>`	`void f();`
`std::function<int()>`	`int f();`
`std::function<void(int)>`	`void f(int i);`
`std::function<void(int,int)>`	`void f(int i, int j);`
`std::function<int(int,int)>`	`int f(int i, int j);`

Vous voyez dans ces exemples que dès que vous connaissez la signature de la fonction, il est relativement simple d'écrire le `std::function` correspondant.

main.cpp

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

int add(int i, int j) {
    return i + j;
}

int main() {
    std::function<int(int,int)> f = add;
    const auto i = f(1, 2);  // 1 + 2
    std::cout << i << std::endl;
}
```

affiche :

```
3
```

Autres fonctionnalités sur les fonctions

Pour terminer ce chapitre, voici rapidement quelques fonctions utilitaires de la bibliothèque standard pour manipuler les fonctions.

Vous avez vu précédemment la fonction `std::invoke` (C++17), pour appeler une fonction avec des arguments.

```cpp
void f(int, std::string);
std::invoke(f, 123, "hello");
```

La fonction `std::apply` (C++17) est assez proche, sauf qu'elle prend un `std::tuple` et applique ses valeurs sur une fonction.

```cpp
void f(int, std::string);
std::apply(f, std::make_tuple(123, "hello"));
```

std::bind

La dernière fonction est `std::bind`. Cette fonction permet de créer une nouvelle fonction à partir d'une fonction, en modifiant les paramètres (par exemple en donnant une valeur à un paramètre ou en modifiant l'ordre des paramètres).

La fonction `std::bind` prend en paramètre une fonction et une liste de paramètres pour appeler cette fonction. La liste des valeurs doit correspondre à la fonction appelée. Par exemple, si une fonction prend quatre paramètres, `std::bind` prendra en argument une fonction et quatre paramètres.

```
void f(int, int, int, int);

auto g = std::bind(f, 1, 2, 3, 4);
```

La fonction `g` définie ci-dessus ne représente pas le résultat de l'appel de la fonction `f` (comme ce serait le cas avec `std::invoke`) mais une nouvelle fonction qui appelle `f` avec ces valeurs. Cela revient à définir la fonction `g` de la façon suivante :

```
void g() {
    f(1, 2, 3, 4);
}
```

Pour appeler la fonction `g`, il faut donc l'appeler sans paramètre :

```
g();
```

Il est possible de créer une fonction qui prend des paramètres de fonction. Pour cela, il faut utiliser des *placeholders* qui représentent les paramètres de la nouvelle fonction. Ces *placeholders* s'écrivent `_1`, `_2`, etc. et correspondent au premier paramètre, au second paramètre, etc. Les *placeholders* sont définis dans l'espace de noms `std::placeholders`.

Par exemple, pour écrire une fonction `g` qui appelle directement la fonction `f` :

```
void f(int, int, int, int);

using namespace std::placeholders;
auto g = std::bind(f, _1, _2, _3, _4);

// est équivalent à :

void g(int i1, int i2, int i3, int i4) {
    f(i1, i2, i3, i4);
}
```

Il est possible de changer l'ordre des paramètres :

```
void f(int, int, int, int);

using namespace std::placeholders;
auto g = std::bind(f, _4, _3, _2, _1);

// est équivalent à :

void g(int i1, int i2, int i3, int i4) {
    f(i4, i3, i2, i1);
}
```

Ou de mélanger des paramètres et des valeurs :

```
void f(int, int, int, int);

using namespace std::placeholders;
auto g = std::bind(f, _2, 12, _1, -21);

// est équivalent à :

void g(int i1, int i2) {
    f(i2, 12, i1, -21);
}
```

Il faut bien faire attention à l'ordre des paramètres dans `std::bind` (qui correspond à l'ordre des paramètres dans la fonction appelée) et l'ordre des *placeholders* (qui correspond à l'ordre des paramètres dans la fonction créée).

Il est également possible de créer des référence sur des variables en utilisant `std::ref` (référence non constante) et `std::cref` (référence constante).

```
void f(int);

int i { 123 };
auto g = std::bind(f, std::ref(i));

// est équivalent à :

void g(int & i) {
    f(i);
}
```

En pratique, `std::bind` est dépréciée par rapport aux fonctions lambdas. Il sera généralement plus simple et lisible d'utiliser les fonctions lambdas.

```
void f(int, int, int, int);

using namespace std::placeholders;
auto g = std::bind(f, _2, 12, _1, -21);

// est équivalent à :

auto g = [](int i1, int i2) { f(i2, 12, i1, -21); };
```


Chapitre précédent	Sommaire principal	Chapitre suivant