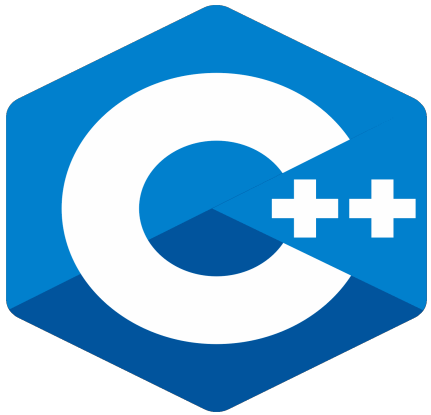
C++ : expression lambda

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



Plan

C++

Expression lambda ?

- Introduite dans **C++11**
- Créée directement dans le code
- Facilitant l'écriture de fonction `inline`
- Souvent déclarée comme paramètre d'une autre fonction/méthode

C++

Syntaxe

```
[ mode_capture ] (paramètres) -> type_retour
{
    // code de la fonction
}
```

C++

Exemple

```
auto lambda = [] () -> void
{ cout << "Hello world" << endl; };
```

C++

Exemple

```
auto lambda = [] () -> void
{ cout << "Hello world" << endl; };
```

Explication

- La fonction ne prend pas de paramètre.
- Elle ne retourne pas de valeur : le type de valeur de retour peut être supprimé et ajouté à la compilation par le compilateur.

C++

Exemple

```
auto lambda = [] () -> void
{ cout << "Hello world" << endl; };
```

Explication

- La fonction ne prend pas de paramètre.
- Elle ne retourne pas de valeur : le type de valeur de retour peut être supprimé et ajouté à la compilation par le compilateur.

On appelle une fonction lambda comme toute autre fonction en C++

```
lambda();
// affiche Hello world
```

C++

Une expression lambda peut prendre un ou plusieurs paramètres

```
auto lambda = [](string nom)
{ cout << "Hello " << nom << endl; };
lambda("wick");
```

L'appel

```
lambda("wick");
// affiche Hello wick
```

C++

Une expression lambda peut accepter des paramètres génériques [C++14]

```
auto lambda = [](auto x, auto y)
{ return x + y; };
```

L'appel

```
auto resultat = lambda(3, 4);
cout << resultat << endl;
// affiche 7

auto resultat2 = lambda(3.2, 5.8);
cout << resultat2 << endl;
// affiche 9
```

C++

Une expression lambda peut accéder à une variable définie dans le contexte global

```
string nom = "wick";
auto lambda = [=]()
{ cout << "Hello " << nom << endl; };
```

L'appel

```
lambda();
// affiche Hello wick
```

C++

Une expression lambda peut accéder à une variable définie dans le contexte global

```
string nom = "wick";
auto lambda = [=]()
{ cout << "Hello " << nom << endl; };
```

L'appel

```
lambda();
// affiche Hello wick
```

Explication

`[=]` : `nom` est passé par valeur.

C++

Une deuxième écriture équivalente

```
string nom = "wick";
auto lambda = [nom]()
{ cout << "Hello " << nom << endl; };
```

L'appel

```
lambda();
// affiche Hello wick
```

C++

Une deuxième écriture équivalente

```
string nom = "wick";
auto lambda = [nom]()
{ cout << "Hello " << nom << endl; };
```

L'appel

```
lambda();
// affiche Hello wick
```


Explication

`[nom]` : `nom` est passé par valeur.

C++

Pour un passage par référence

```
string nom = "wick";
auto lambda = [&]()
{ nom[0] = nom[0] + ('A' - 'a'); };
```

L'appel

```
lambda();
cout << nom << endl;
// affiche Wick
```

C++

Pour un passage par référence

```
string nom = "wick";
auto lambda = [&]()
{ nom[0] = nom[0] + ('A' - 'a'); };
```

L'appel

```
lambda();
cout << nom << endl;
// affiche Wick
```


Explication

`[&]` : `nom` est passé par référence.

C++

Pour un passage par référence et un deuxième par valeur (copie)

```
int i = 5, j = 2;
auto lambda = [&i, j]()
{ i += j; };
```

L'appel

```
lambda();
cout << i << " " << j << endl;
// affiche 7 2
```

C++

On peut aussi spécifier un mode de capture par défaut (sans préciser de variable)

```
int i = 5, j = 2;
auto lambda = [&, j]()
{ i += j; };
```

C++

On peut aussi spécifier un mode de capture par défaut (sans préciser de variable)

```
int i = 5, j = 2;
auto lambda = [&, j]()
{ i += j; };
```

[&, j]

- `&` : le mode de capture par défaut est par référence.
- `j` : le mode de capture pour `j` est par valeur.

C++

On peut aussi spécifier un mode de capture par défaut (sans préciser de variable)

```
int i = 5, j = 2;
auto lambda = [&, j]()
{ i += j; };
```


[&, j]

- `&` : le mode de capture par défaut est par référence.
- `j` : le mode de capture pour `j` est par valeur.

L'appel

```
lambda();
cout << i << " " << j << endl;
// affiche 7 2
```

C++

Considérons le vecteur suivant

```
vector<int> vecteur{2, 5, 8, 4};
```

C++

Considérons le vecteur suivant

```
vector<int> vecteur{2, 5, 8, 4};
```

N'oublions pas les `include` suivants

```
#include <vector>
#include <algorithm>
```

C++

Pour compter le nombre d'élément d'un vecteur qui respectent une condition

```
int n = count_if(vecteur.begin(), vecteur.end(),
                 [=](int elt)
                 { return (elt >= 5); });

cout << n << endl;
// affiche 2
```

C++

`find_if` **retourne un itérateur sur le premier élément qui respecte la condition**

```
auto n = find_if(vecteur.begin(), vecteur.end(),
                 [=](int elt)
                 { return (elt > 5); });

cout << *n << endl;
// affiche 8
```

C++

Pour vérifier si tous éléments d'un vecteur respectent une condition

```
auto b = all_of(vecteur.begin(), vecteur.end(), [=](int elt)
                { return (elt > 5); });

cout << boolalpha << b << endl;
// affiche false
```

C++

Pour vérifier si tous éléments d'un vecteur respectent une condition

```
auto b = all_of(vecteur.begin(), vecteur.end(), [=](int elt)
                { return (elt > 5); });

cout << boolalpha << b << endl;
// affiche false
```

Pour vérifier si au moins un élément d'un vecteur respecte une condition

```
auto b = any_of(vecteur.begin(), vecteur.end(), [=](int elt)
                { return (elt > 5); });

cout << boolalpha << b << endl;
// affiche true
```

C++

Pour vérifier si tous éléments d'un vecteur respectent une condition

```
auto b = all_of(vecteur.begin(), vecteur.end(), [=](int elt)
                { return (elt > 5); });

cout << boolalpha << b << endl;
// affiche false
```

Pour vérifier si au moins un élément d'un vecteur respecte une condition

```
auto b = any_of(vecteur.begin(), vecteur.end(), [=](int elt)
                { return (elt > 5); });

cout << boolalpha << b << endl;
// affiche true
```

Pour vérifier si aucun élément d'un vecteur respecte une condition

```
auto b = none_of(vecteur.begin(), vecteur.end(), [=](int elt)
                { return (elt > 5); });

cout << boolalpha << b << endl;
// affiche false
```

C++

Pour parcourir un vecteur

```
vector<int> vecteur{2, 5, 8, 4};
for_each(vecteur.begin(), vecteur.end(), [=](int elt)
                    { cout << elt << " "; });

// affiche 2 5 8 4
```

C++

Pour appliquer une transformation sur chaque élément du vecteur

```
transform(vecteur.begin(), vecteur.end(), vecteur.begin(),
          [](int elt) -> int
          { return (elt + 1); });

for_each(vecteur.begin(), vecteur.end(), [](int elt)
         { cout << elt << " "; });
// affiche 3 6 9 5
```

C++

Pour appliquer une transformation sur chaque élément du vecteur

```
transform(vecteur.begin(), vecteur.end(), vecteur.begin(),
          [](int elt) -> int
          { return (elt + 1); });

for_each(vecteur.begin(), vecteur.end(), [](int elt)
         { cout << elt << " "; });
// affiche 3 6 9 5
```

`transform` **peut aussi accepter un opérateur binaire**

```
vector<int> vecteur{2, 5, 8, 4};
vector<int> vecteur2{1, 0, 9, 6};
vector<int> result{};

result.resize(vecteur.size());

transform(vecteur.begin(), vecteur.end(), vecteur2.begin(), result.begin(),
          [](int a, int b) -> int
          { return a + b; });

for_each(result.begin(), result.end(), [](int elt)
         { cout << elt << " "; });
// affiche 3 5 17 10
```

C++

Remarque

Les fonctions lambda pourront aussi être remplacées par des fonctions de la librairie `functional` (`plus`, `multiplies`...) ou des functors.

C++

Remarque

Les fonctions lambda pourront aussi être remplacées par des fonctions de la librairie `functional` (`plus`, `multiplies`...) ou des functors.

Pour consulter la liste de toutes les autres fonctions

`https://en.cppreference.com/w/cpp/utility/functional`