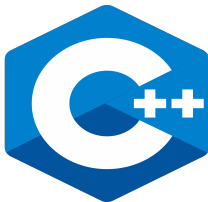


# C++ : expression lambda

**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

## 1 Introduction

## 2 Exemples

- Expression lambda sans paramètres
- Expression lambda avec paramètres
- Expression lambda avec paramètres génériques
- Capture de valeur
- Capture de référence
- Capture mixte
- Mot-clé `mutable`

## 3 Expression lambda et conteneurs STL

- `count_if`
- `find_if`
- `x_of`
- `for_each`
- `transform`

## Expression lambda : contexte

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

## Expression lambda : structure

- `[]` : capture
- `()` : paramètre
- `{}` : corps de la fonction

## Syntaxe

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

# C++

## Exemple

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

© Achref EL MOUELHI ©

# 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; };
```

## 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++

## Exercice 1 : qu'affiche le programme suivant ?

```
int main()
{
    int i = 5;
    int j;
    auto lambda = [i]()
    { return i + 5; };

    j = lambda();
    cout << j << ", " << i << endl;
    i = 20;
    j = lambda();
    cout << j << ", " << i << endl;
    return 0;
}
```

# C++

## Exercice 1 : qu'affiche le programme suivant ?

```
int main()
{
    int i = 5;
    int j;
    auto lambda = [i]()
    { return i + 5; };

    j = lambda();
    cout << j << ", " << i << endl;
    i = 20;
    j = lambda();
    cout << j << ", " << i << endl;
    return 0;
}
```

### Surprise

```
10, 5
10, 20
```

## 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; };
```

© Achref EL MOUELHI ©

# 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++

Pour pouvoir modifier une variable localement (mais pas globalement), on utilise le mot-clé `mutable`

```
int main()
{
    int i = 5, j = 2;
    auto lambda = [&i, j]() mutable
    { i += j; };
    lambda();
    cout << i << " " << j << endl;
    i = 3;
    lambda();
    cout << i << " " << j << endl;
    return 0;
}
```

# C++

Pour pouvoir modifier une variable localement (mais pas globalement), on utilise le mot-clé `mutable`

```
int main()
{
    int i = 5, j = 2;
    auto lambda = [&i, j]() mutable
    { i += j; };
    lambda();
    cout << i << " " << j << endl;
    i = 3;
    lambda();
    cout << i << " " << j << endl;
    return 0;
}
```

## Résultat

```
7 2
5 2
```



## Considérons le vecteur suivant

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

© Achref EL MOUËL

Considérons le vecteur suivant

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

N'oublions pas les `include` suivants

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

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
```

`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
```

© Achref EL MOUELHI ©

# 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
```

## 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
```

© Achref EL MOUËLHI

# 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
```

## Autres fonctions prenant comme paramètre une expression lambda ?

- `copy_if`
- `reduce`
- `remove_if`
- `replace_if`
- ...

## Autres fonctions prenant comme paramètre une expression lambda ?

- `copy_if`
- `reduce`
- `remove_if`
- `replace_if`
- ...

## Liste complète

<https://en.cppreference.com/w/cpp/algorithm>

## Remarque

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

© Achref EL ME

## 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>