



# INTRODUCTION AUX LAMBDA

## DEFINITION

Les lambdas en C++ sont des fonctions anonymes. Elles permettent de définir des fonctions inline. Utilisées pour des opérations courtes et locales. Introduites dans le standard C++11. Syntaxe générale :

[capture] (parameters) -> return\_type { body }.

# UTILISATION

Les lambdas sont souvent utilisées avec des algorithmes STL. Elles simplifient le code en évitant la déclaration de fonctions séparées. Permettent d'écrire des fonctions locales dans le contexte d'utilisation. Exemple d'utilisation avec `std::for_each`:

```
std::for_each(vec.begin(), vec.end(), [](int n) { std::cout << n << " "; });
```

## AVANTAGES

- Code plus concis et lisible.
- Facilite la programmation fonctionnelle.
- Evite la pollution de l'espace de noms avec des fonctions temporaires.
- Permet de capturer des variables locales.

## COMPARAISON AVEC LES FONCTIONS CLASSIQUES

Caractéristique	Lambdas	Fonctions classiques
Nom	Anonyme	Nommée
Localité	Locale	Globale ou locale
Captures	Variables locales	Non applicable
Syntaxe	<code>[capture](params){}</code>	<code>type nom(params){}</code>

## LIMITES

- Peut être moins lisible si mal utilisée.
- Limitées à des opérations courtes et simples.
- Moins flexibles que les fonctions classiques pour des opérations complexes.
- Peut introduire des overheads si mal optimisées.



## CONTEXTE D'UTILISATION

- Utilisées dans les algorithmes STL comme `std::for_each`, `std::transform`.
- Utiles pour des callbacks et des handlers d'événements.
- Pratiques pour des opérations locales et temporaires.
- Souvent utilisées dans les programmes modernes C++ pour améliorer la lisibilité.

## HISTORIQUE ET EVOLUTION

- Introduites dans le standard C++11.
- Améliorations et extensions dans C++14 et C++17.
- C++14 a introduit les lambdas généralisées (generic lambdas).
- C++20 a ajouté des fonctionnalités supplémentaires comme les lambdas constexpr.

# **SYNTAXE DES LAMBDAS**

## **DEFINITION**

Les lambdas en C++ sont des fonctions anonymes. Elles permettent de définir des fonctions directement dans le code. Utilisées pour des opérations simples et locales. Introduites dans C++11 pour simplifier le code.

# SYNTAXE DE BASE

La syntaxe de base d'une lambda est la suivante :

```
[]() { /* code */ }
```

Les crochets [ ] délimitent la capture. Les parenthèses ( ) contiennent les paramètres. Les accolades {} contiennent le corps de la lambda.

# PARAMÈTRES

Les lambdas peuvent accepter des paramètres :

```
[](int a, int b) { return a + b; }
```

Les paramètres sont définis entre les parenthèses ( ). Comme pour les fonctions normales.

## RETOUR DE VALEUR

Le type de retour peut être spécifié :

```
[](int a, int b) -> int { return a + b; }
```

Utilisez `->` pour spécifier le type de retour. Sinon, le type de retour est déduit automatiquement.

## UTILISATION AVEC STD::FUNCTION

Les lambdas peuvent être utilisées avec `std::function`:

```
std::function<int(int, int)> add = [](int a, int b) { return a + b; };
```

`std::function` permet de stocker et d'utiliser les lambdas.

## EXAMPLE SIMPLE

Un exemple simple d'utilisation de lambda :

```
auto add = [](int a, int b) { return a + b; };
int result = add(2, 3); // result vaut 5
```

La lambda `add` additionne deux nombres.

## **COMPARAISON AVEC LES FONCTIONS NORMALES**

Les lambdas sont plus concises que les fonctions normales. Elles sont définies directement dans le code. Pas besoin de déclarations séparées. Idéales pour des opérations simples et locales.

# CAPTURE DE VARIABLES

## DEFINITION

La capture de variables dans une lambda permet d'utiliser des variables externes à l'intérieur de la lambda. Les variables peuvent être capturées par valeur, par référence ou de manière implicite. La capture se fait à l'aide de crochets [ ] dans la syntaxe de la lambda.

## CAPTURE PAR VALEUR

La capture par valeur copie la valeur des variables externes dans la lambda. Syntaxe :

```
[variable](int a, int b) { return a + variable; }
```

Les modifications à l'intérieur de la lambda n'affectent pas les variables externes.

## CAPTURE PAR RÉFÉRENCE

La capture par référence permet à la lambda de modifier les variables externes. Syntaxe :

```
[&variable](int a, int b) { variable += a + b; }
```

Les modifications à l'intérieur de la lambda affectent directement les variables externes.

## CAPTURE IMPLICITE

La capture implicite capture toutes les variables nécessaires. Capture par valeur :

```
[=](int a, int b) { return a + variable; }
```

Capture par référence :

```
[&](int a, int b) { variable += a + b; }
```

# CAPTURE EXPLICITE

La capture explicite permet de choisir quelles variables capturer. Par valeur :

```
[variable](int a, int b) { return a + variable; }
```

Par référence :

```
[&variable](int a, int b) { variable += a + b; }
```

## EXEMPLES DE CAPTURE

Capture par valeur :

```
int x = 10;  
auto lambda = [x]() { return x; };
```

Capture par référence :

```
int x = 10;  
auto lambda = [&x]() { x += 5; };
```



## **AVANTAGES**

- Flexibilité dans l'utilisation des variables externes.
- Permet des expressions plus concises.

## INCONVÉNIENTS

- Peut entraîner des erreurs si mal utilisé.
- La capture par référence peut causer des effets de bord inattendus.

# **EXPRESSIONS DE RETOUR**

## DÉFINITION

Les expressions de retour dans les lambdas permettent de spécifier explicitement le type de retour. Elles sont utiles pour clarifier le type de valeur renvoyée par la lambda. Cela peut être nécessaire lorsque le type de retour n'est pas évident.

## SYNTAXE DES EXPRESSIONS DE RETOUR

La syntaxe pour spécifier une expression de retour est la suivante :

```
auto lambda = []() -> type {  
    // corps de la lambda  
};
```

## EXAMPLE SIMPLE

Voici un exemple simple d'utilisation d'une expression de retour :

```
auto add = [](int a, int b) -> int {
    return a + b;
};
```

## **TYPES DE RETOUR**

Les types de retour peuvent être :

- Types primitifs (int, float, etc.)
- Types complexes (struct, class, etc.)
- Pointeurs et références

# UTILISATION AVEC DES FONCTIONS

Les lambdas avec expressions de retour peuvent être passées à des fonctions :

```
void applyLambda(auto lambda) {
    std::cout << lambda(2, 3);
}

applyLambda([](int a, int b) -> int {
    return a + b;
});
```

## AVANTAGES DES EXPRESSIONS DE RETOUR

- Clarifient le type de retour
- Évitent les erreurs de type
- Améliorent la lisibilité du code
- Facilitent le débogage

## LIMITATIONS DES EXPRESSIONS DE RETOUR

- Peuvent rendre le code plus verbeux
- Nécessitent une syntaxe supplémentaire
- Peuvent être redondantes si le type de retour est évident

# FONCTIONS ANONYMES

## DÉFINITION

Les lambdas sont des fonctions anonymes. Introduites dans C++11. Permettent de définir des fonctions inline. Utilisées pour des opérations courtes et locales. Syntaxe compacte et facile à lire. Très utiles avec les algorithmes de la STL.

# SYNTAXE

Syntaxe de base des lambdas :

```
[ captures ] ( paramètres ) -> type_retour { corps }
```

Exemple :

```
auto lambda = []() { return 42; };
```

# CAPTURES

Les lambdas peuvent capturer des variables de leur scope. Syntaxe des captures :

```
[ capture1, capture2 ]
```

Types de captures :

- Par valeur : [=]
- Par référence : [&]

# UTILISATION DE VARIABLES LOCALES

Capturer des variables locales par valeur :

```
int x = 10;  
auto lambda = [x]() { return x; };
```

Capturer des variables locales par référence :

```
int x = 10;  
auto lambda = [&x]() { x = 20; };
```

## UTILISATION DE VARIABLES GLOBALES

Les lambdas peuvent accéder aux variables globales. Pas besoin de les capturer explicitement.

```
int globalVar = 5;  
auto lambda = []() { return globalVar; };
```

# EXEMPLES SIMPLES

Lambda sans capture, sans paramètres :

```
auto lambda = []() { return 42; };
```

Lambda avec capture et paramètres :

```
int x = 10;
auto lambda = [x](int y) { return x + y; };
```

# COMPARAISON AVEC LES FONCTIONS NORMALES

Fonction normale :

```
int add(int a, int b) { return a + b; }
```

Lambda équivalente :

```
auto add = [](int a, int b) { return a + b; };
```



## AVANTAGES

- Syntaxe concise
- Facile à utiliser avec la STL
- Captures automatiques

## INCONVÉNIENTS

- Moins lisibles pour des fonctions complexes
- Pas de nom explicite pour la fonction

# **UTILISATION DES LAMBDAS AVEC LES ALGORITHMES DE LA STL**

## **DEFINITION**

Les lambdas en C++ sont des fonctions anonymes définies directement dans le code. Elles peuvent capturer des variables de leur contexte d'exécution. Utilisées pour des opérations courtes et locales. Syntaxe concise et flexible. Introduites dans C++11.

# QUAND L'UTILISER

Utilisez les lambdas pour :

- Les opérations courtes et locales.
- Les callbacks.
- Les algorithmes de la STL.
- Remplacer les petites fonctions membres.
- Simplifier le code.

# SYNTAXE

La syntaxe de base d'une lambda en C++ est :

```
[ capture ] ( params ) -> ret { body }
```

- **capture** : Variables capturées.
- **params** : Paramètres de la lambda.
- **ret** : Type de retour (optionnel).
- **body** : Corps de la fonction.

## EXAMPLE

Déclaration et utilisation d'une lambda simple :

```
auto add = [](int a, int b) { return a + b; };
int result = add(3, 4); // result = 7
```

## UTILISATION AVEC STD::FOR\_EACH

Utilisation d'une lambda avec std::for\_each :

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

std::vector<int> vec = {1, 2, 3, 4, 5};
std::for_each(vec.begin(), vec.end(), [](&n) { n *= 2; });
```

# UTILISATION AVEC STD::TRANSFORM

Utilisation d'une lambda avec `std::transform`:

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

std::vector<int> vec = {1, 2, 3, 4, 5};
std::vector<int> result(vec.size());
std::transform(vec.begin(), vec.end(), result.begin(), [] (int n) { return n * n; });
```

# UTILISATION AVEC STD::SORT

Utilisation d'une lambda avec std::sort :

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

std::vector<int> vec = {5, 2, 8, 1, 3};
std::sort(vec.begin(), vec.end(), [](int a, int b) { return a < b; });
```

## UTILISATION AVEC STD::FIND\_IF

Utilisation d'une lambda avec std::find\_if :

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

std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = std::find_if(vec.begin(), vec.end(), [] (int n) { return n > 3; });
if (it != vec.end()) {
    std::cout << "Found: " << *it << std::endl;
}
```



## AVANTAGES

- Code plus concis et lisible.
- Simplifie les callbacks et les fonctions locales.
- Flexibilité dans les algorithmes STL.

## **INCONVENIENTS**

- Peut être difficile à lire pour les débutants.
- Mauvaise utilisation peut rendre le code complexe.
- Capture de variables peut entraîner des erreurs subtiles.

# LAMBDA ET FONCTIONS MEMBRES

## DEFINITION

Les lambdas en C++ sont des fonctions anonymes. Elles permettent de définir des fonctions à la volée. Elles peuvent capturer des variables de leur environnement. Les lambdas peuvent être utilisées comme des fonctions normales. Elles sont souvent utilisées avec les algorithmes de la STL. Les lambdas peuvent être assignées à des variables.

# SYNTAXE

La syntaxe de base d'une lambda en C++ est la suivante :

```
auto lambda = []() { /* code */ };
```

Les lambdas peuvent prendre des paramètres :

```
auto lambda = [](int x) { return x * 2; };
```

Elles peuvent aussi retourner des valeurs.

## EXEMPLE BASIQUE

Définition et utilisation d'une lambda :

```
auto add = [](int a, int b) { return a + b; };
int result = add(3, 4); // result vaut 7
```

Les lambdas peuvent être appelées comme des fonctions normales.

## UTILISATION AVEC DES FONCTIONS MEMBRES

Les lambdas peuvent être utilisées comme des fonctions membres. Elles peuvent accéder aux membres de la classe.

```
class MyClass {
public:
    void myMethod() {
        auto lambda = [this]() { /* code utilisant les membres */ };
        lambda();
    }
};
```

# CAPTURES IMPLICITES ET EXPLICITES

Les lambdas peuvent capturer des variables de leur environnement. Capture implicite :

```
int x = 10;  
auto lambda = [=]() { return x * 2; };
```

Capture explicite :

```
int x = 10;  
auto lambda = [&x]() { x *= 2; };
```



## AVANTAGES

- Syntaxe concise
- Flexibilité
- Intégration avec la STL

## **INCONVÉNIENTS**

- Peut rendre le code moins lisible
- Complexité de la gestion des captures

# LAMBDA ET PORTÉE DES VARIABLES

## DÉFINITION

Les lambdas en C++ sont des fonctions anonymes. Elles peuvent capturer et utiliser des variables du scope environnant. Définies avec la syntaxe [capture] (params) { body }. Utiles pour des fonctions courtes et locales. Permettent de passer des fonctions comme arguments.

## **PORTEE DES VARIABLES LOCALES**

Les variables locales sont définies à l'intérieur d'une fonction. Elles ne sont accessibles qu'à l'intérieur de cette fonction. Les lambdas peuvent capturer ces variables. La capture permet de les utiliser dans le corps de la lambda.

## **PORTÉE DES VARIABLES GLOBALES**

Les variables globales sont définies en dehors de toutes fonctions. Elles sont accessibles partout dans le programme. Les lambdas peuvent directement accéder aux variables globales. Pas besoin de capture pour les utiliser dans les lambdas.

## CAPTURE PAR VALEUR

La capture par valeur copie la variable capturée. Syntaxe : [=] ou [var]. Les modifications dans la lambda n'affectent pas l'original. Utile pour éviter les effets de bord.

## CAPTURE PAR RÉFÉRENCE

La capture par référence utilise la variable capturée directement. Syntaxe : [&] ou [&var]. Les modifications dans la lambda affectent l'original. Utile pour modifier des variables externes.

## CAPTURE IMPLICITE

La capture implicite utilise toutes les variables nécessaires. Syntaxe : [=] pour capture par valeur. Syntaxe : [&] pour capture par référence. Pratique mais peut rendre le code moins clair.

## CAPTURE EXPLICITE

La capture explicite spécifie chaque variable capturée. Syntaxe : `[var1, &var2]`. Plus clair et évite les captures non intentionnelles. Recommandé pour un code plus lisible et maintenable.

# EXEMPLES PRATIQUES

```
int x = 10;
auto lambda = [x]() { return x + 1; };
std::cout << lambda(); // Affiche 11
```

```
int x = 10;
auto lambda = [&x]() { x += 1; };
lambda();
std::cout << x; // Affiche 11
```



## **AVANTAGES**

- Code concis et lisible.
- Facilite la programmation fonctionnelle.
- Capture de variables locales.

## INCONVÉNIENTS

- Peut rendre le code complexe.
- Capture implicite peut causer des erreurs.
- Moins performant que les fonctions membres.

# LAMBDA ET PERFORMANCE

# DEFINITION

Les lambdas en C++ sont des fonctions anonymes définies à la volée. Elles permettent de créer des fonctions locales sans nom. Syntaxe de base :

```
auto lambda = []() { /* code */ };
```

Les lambdas peuvent capturer des variables locales.

## IMPACT SUR LA PERFORMANCE

Les lambdas peuvent améliorer la performance en réduisant l'overhead. Elles permettent d'éviter les appels de fonctions classiques. Les lambdas peuvent être en ligne, réduisant le coût d'appel. Cependant, l'utilisation excessive peut augmenter la complexité du code. Analysez toujours l'impact sur la performance avant d'utiliser des lambdas.

## COMPARAISON AVEC FONCTIONS CLASSIQUES

Critère	Lambdas	Fonctions classiques
Définition	Anonyme, locale	Nommée, globale ou locale
Overhead	Réduit	Peut être élevé
Flexibilité	Haute, captures locales	Moins flexible
Lisibilité	Peut diminuer	Généralement meilleure

## **OPTIMISATION DES LAMBDA**

Utilisez des lambdas capturant par référence pour réduire la copie. Privilégiez les lambdas en ligne pour éviter les appels de fonctions. Évitez les captures inutiles pour réduire la taille de la lambda. Utilisez des lambdas simples pour maintenir la lisibilité du code. Analysez les performances avant et après l'optimisation.

## CAS D'UTILISATION

- Algorithmes STL (e.g., `std::for_each`, `std::transform`)
- Callbacks dans les interfaces graphiques
- Fonctions temporaires dans les tests unitaires
- Code nécessitant des fonctions locales simples
- Remplacement de petites fonctions globales

## EXEMPLES DE PERFORMANCE

Comparaison d'une boucle avec et sans lambda :

```
std::vector<int> vec = {1, 2, 3, 4, 5};

// Avec lambda
std::for_each(vec.begin(), vec.end(), [](int &n){ n *= 2; });

// Sans lambda
for (auto &n : vec) { n *= 2; }
```

Analysez le temps d'exécution pour mesurer les gains.

## MEILLEURES PRATIQUES

- Utilisez des lambdas pour des tâches simples et locales.
- Évitez les captures inutiles pour optimiser la performance.
- Préférez les lambdas en ligne pour réduire l'overhead.
- Maintenez la lisibilité en évitant les lambdas complexes.
- Testez toujours les performances avant d'adopter une solution.

# LAMBDAS ET THREADS

## **DEFINITION**

Les lambdas en C++ sont des fonctions anonymes. Elles permettent de définir des fonctions de manière concise. Les lambdas peuvent capturer des variables du scope environnant. Elles sont utiles pour les opérations simples et les callbacks. Les lambdas sont souvent utilisées en programmation parallèle.

## UTILISATION AVEC THREADS

Les lambdas peuvent être passées comme arguments aux threads. Cela permet d'exécuter du code en parallèle facilement. Les lambdas facilitent la création de threads légers. Elles permettent de capturer des variables locales pour les threads. L'utilisation de lambdas avec `std::thread` est courante en C++.

# SYNTAXE

La syntaxe de base d'une lambda en C++ est la suivante :

```
auto lambda = []() {  
    // Code de la lambda  
};
```

Pour l'utiliser avec un thread :

```
std::thread t(lambda);
```

# CAPTURES DANS LES LAMBDAS

Les lambdas peuvent capturer des variables de plusieurs façons :

- Par valeur : [=]
- Par référence : [&]
- Captures spécifiques : [x, &y]

Exemple de capture par valeur et référence :

```
int x = 10;
int y = 20;
auto lambda = [x, &y]() {
    // Utilise x par valeur et y par référence
};
```

# EXEMPLE DE LAMBDA AVEC THREAD

Exemple complet d'utilisation d'une lambda avec un thread :

```
#include <iostream>
#include <thread>

int main() {
    int x = 10;
    std::thread t([x]() {
        std::cout << "La valeur de x est : " << x << std::endl;
    });
    t.join();
    return 0;
}
```



## AVANTAGES

- Syntaxe concise et claire
- Capture de variables locales
- Facilité d'utilisation avec `std::thread`

## **INCONVENIENTS**

- Peut être moins lisible pour des fonctions complexes
- Problèmes potentiels de capture par référence

## MEILLEURES PRATIQUES

- Utiliser des lambdas pour des tâches simples et courtes.
- Préférer la capture par valeur pour éviter les problèmes de synchronisation.
- Utiliser des noms de variables explicites pour améliorer la lisibilité.
- Toujours gérer correctement la durée de vie des variables capturées.
- Tester les lambdas dans des environnements multithread pour éviter les bugs.

# **UTILISATION DE LAMBDA AVEC LES CLASSES DE CONTENEURS**

## **DEFINITION**

Les lambdas en C++ sont des fonctions anonymes. Elles permettent de définir des comportements à la volée. Les lambdas peuvent être utilisées avec des classes de conteneurs. Elles simplifient les opérations sur les éléments des conteneurs.

# SYNTAXE

La syntaxe de base d'une lambda en C++ est :

```
[captures](parameters) -> return_type {  
    // corps de la fonction  
};
```

## EXEMPLE DE LAMBDA AVEC VECTOR

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

std::vector<int> v = {1, 2, 3, 4, 5};
std::for_each(v.begin(), v.end(), [](int &n) { n *= 2; });
for (int n : v) std::cout << n << " "; // Affiche : 2 4 6 8 10
```

## EXEMPLE DE LAMBDA AVEC MAP

```
#include <map>
#include <algorithm>
#include <iostream>

std::map<int, int> m = {{1, 2}, {3, 4}, {5, 6}};
std::for_each(m.begin(), m.end(), [](std::pair<int, int> &p) { p.second *= 2; });
for (const auto &p : m) std::cout << p.first << ":" << p.second << " "; // Affiche : 1:4 3:8 5:12
```

# UTILISATION AVEC FIND\_IF

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

std::vector<int> v = {1, 2, 3, 4, 5};
auto it = std::find_if(v.begin(), v.end(), [] (int n) { return n > 3; });
if (it != v.end()) std::cout << "Trouvé : " << *it << std::endl; // Affiche : Trouvé : 4
```

# UTILISATION AVEC FOR\_EACH

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

std::vector<int> v = {1, 2, 3, 4, 5};
std::for_each(v.begin(), v.end(), [] (int &n) { n += 1; });
for (int n : v) std::cout << n << " "; // Affiche : 2 3 4 5 6
```

# UTILISATION AVEC TRANSFORM

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

std::vector<int> v = {1, 2, 3, 4, 5};
std::vector<int> result(v.size());
std::transform(v.begin(), v.end(), result.begin(), [](int n) { return n * n; });
for (int n : result) std::cout << n << " "; // Affiche : 1 4 9 16 25
```

## CAPTURES ET CONTENEURS

Les lambdas peuvent capturer des variables locales. Les captures se font par valeur ou par référence.  
Exemple de capture par valeur :

```
int factor = 2;
std::for_each(v.begin(), v.end(), [factor](int &n) { n *= factor; });
```

## CAPTURES ET CONTENEURS (SUITE)

Exemple de capture par référence :

```
int sum = 0;
std::for_each(v.begin(), v.end(), [&sum](int n) { sum += n; });
std::cout << "Somme : " << sum << std::endl; // Affiche : Somme : 15
```



## **AVANTAGES**

- Code plus concis et lisible.
- Flexibilité pour les opérations sur les conteneurs.
- Facilite l'utilisation de fonctions standard de la STL.

## **INCONVÉNIENTS**

- Peut rendre le code moins compréhensible pour les débutants.
- Les captures peuvent entraîner des erreurs subtiles.
- Difficulté à déboguer les lambdas complexes.

# **UTILISATION DE LAMBDAS AVEC LES POINTEURS INTELLIGENTS**

## DÉFINITION DES LAMBDA

Les lambdas sont des fonctions anonymes définies directement dans le code. Elles permettent de créer des fonctions courtes et concises. Elles sont souvent utilisées pour des opérations de courte durée. Les lambdas peuvent capturer des variables de leur environnement. Elles sont définies à l'aide de la syntaxe [ ].

# SYNTAXE DES LAMBDA

La syntaxe d'une lambda en C++ est la suivante :

```
auto lambda = []() {
    // Code de la lambda
};
```

Les lambdas peuvent prendre des paramètres et retourner des valeurs. Exemple avec paramètres et retour :

```
auto lambda = [](int a, int b) -> int {
    return a + b;
};
```

# INTRODUCTION AUX POINTEURS INTELLIGENTS

Les pointeurs intelligents gèrent automatiquement la durée de vie des objets. Ils aident à éviter les fuites de mémoire. Types principaux : `std::shared_ptr`, `std::unique_ptr`, `std::weak_ptr`. Ils sont définis dans la bibliothèque `<memory>`.

## UTILISATION DE LAMBDAS AVEC STD::SHARED\_PTR

`std::shared_ptr` permet le partage de la propriété d'un objet. Les lambdas peuvent être utilisées avec `std::shared_ptr` pour des opérations comme :

```
std::shared_ptr<int> sp = std::make_shared<int>(10);
auto lambda = [sp]() {
    std::cout << *sp << std::endl;
};
lambda();
```

## UTILISATION DE LAMBDAS AVEC STD::UNIQUE\_PTR

`std::unique_ptr` assure la propriété unique d'un objet. Les lambdas peuvent capturer un `std::unique_ptr` par référence :

```
std::unique_ptr<int> up = std::make_unique<int>(20);
auto lambda = [&up]() {
    std::cout << *up << std::endl;
};
lambda();
```

## UTILISATION DE LAMBDAS AVEC STD::WEAK\_PTR

`std::weak_ptr` fournit une référence non-propriétaire à un objet géré par `std::shared_ptr`. Les lambdas peuvent vérifier la validité de `std::weak_ptr`:

```
std::shared_ptr<int> sp = std::make_shared<int>(30);
std::weak_ptr<int> wp = sp;
auto lambda = [wp]() {
    if (auto sp_locked = wp.lock()) {
        std::cout << *sp_locked << std::endl;
    }
};
lambda();
```

# EXEMPLES PRATIQUES

Exemple avec `std::shared_ptr` et lambda :

```
std::shared_ptr<int> sp = std::make_shared<int>(40);
auto print = [sp]() {
    std::cout << "Value: " << *sp << std::endl;
};
print();
```

Exemple avec `std::unique_ptr` et lambda :

```
std::unique_ptr<int> up = std::make_unique<int>(50);
auto print = [&up]() {
    std::cout << "Value: " << *up << std::endl;
};
print();
```



## AVANTAGES

- Simplifient le code en évitant les déclarations de fonctions séparées.
- Facilitent la gestion des ressources avec les pointeurs intelligents.
- Capturent facilement les variables nécessaires.

## **INCONVÉNIENTS**

- Peuvent rendre le code moins lisible si abusées.
- Les captures par valeur peuvent causer des copies inutiles.
- Les captures par référence nécessitent une gestion attentive de la durée de vie des objets.

# **COMPARAISON AVEC LES FONCTIONS CLASSIQUES**

## **DEFINITION**

Les lambdas sont des fonctions anonymes définies directement dans le code. Elles permettent de créer des fonctions sans les nommer. Les lambdas sont souvent utilisées pour des tâches simples et temporaires. Elles sont définies à l'aide de la syntaxe [ ] () {}.

# SYNTAXE

La syntaxe d'une lambda en C++ est la suivante :

```
auto lambda = []() {  
    // Corps de la fonction  
};
```

## SIMPLICITÉ

Les lambdas simplifient le code en réduisant le besoin de déclarer des fonctions séparées. Elles sont utiles pour les fonctions courtes et les callbacks. Moins de code boilerplate comparé aux fonctions classiques.

## FLEXIBILITÉ

Les lambdas peuvent capturer des variables de leur environnement. Elles peuvent être utilisées comme arguments de fonctions. Elles peuvent être assignées à des variables ou des pointeurs de fonctions.

## **PERFORMANCE**

Les lambdas peuvent parfois être plus performantes que les fonctions classiques. Elles peuvent éviter les surcoûts liés aux appels de fonctions. Cependant, leur performance dépend du compilateur et de l'utilisation spécifique.





## **COMPARAISON DE LA LISIBILITÉ**

Les lambdas rendent le code plus concis et lisible dans certains cas. Les fonctions classiques peuvent être plus lisibles pour des tâches complexes. La lisibilité dépend souvent du contexte et des préférences du développeur.

## **COMPARAISON DE LA MAINTENANCE**

Les lambdas peuvent être plus difficiles à maintenir si elles deviennent trop complexes. Les fonctions classiques sont plus faciles à documenter et à tester séparément. Il est important de choisir l'approche la plus adaptée à la complexité de la tâche.

## CAS D'UTILISATION ADAPTÉS

Les lambdas sont idéales pour :

- Les callbacks
- Les fonctions courtes et simples
- Les algorithmes de bibliothèque standard (STL) Les fonctions classiques sont préférables pour :
- Les tâches complexes
- Les fonctions réutilisables
- Le code nécessitant une documentation détaillée

# LIMITATIONS DES LAMBDAS

## **PORTÉE LIMITÉE**

Les lambdas ont une portée limitée au bloc où elles sont définies. Elles ne peuvent pas être utilisées en dehors de ce bloc. Cela peut limiter leur réutilisabilité et leur flexibilité.

## PAS DE SURCHARGE

Les lambdas ne peuvent pas être surchargées. Cela signifie que vous ne pouvez pas définir plusieurs lambdas avec le même nom mais des signatures différentes. Cela limite leur utilisation dans certains contextes.

## IMPOSSIBILITÉ DE RÉCURSIVITÉ

Les lambdas en C++ ne peuvent pas être récursives. Elles ne peuvent pas s'appeler elles-mêmes directement. Pour des fonctions récursives, il faut utiliser des fonctions classiques.

## **PERFORMANCE**

Les lambdas peuvent parfois être moins performantes que les fonctions classiques. Cela dépend de la complexité de la capture et de l'utilisation. Il est important de tester et d'optimiser si nécessaire.

## **COMPLEXITÉ DE LECTURE**

Les lambdas peuvent être difficiles à lire et à comprendre. Surtout si elles sont imbriquées ou très complexes. Il est important de les utiliser avec parcimonie et de documenter leur utilisation.

## LIMITES DE CAPTURE

Les lambdas ne peuvent capturer que les variables locales. Les variables globales ou statiques ne peuvent pas être capturées. Cela peut limiter leur utilisation dans certains cas.

## **RESTRICTIONS SUR LES TYPES DE RETOUR**

Les lambdas ont des restrictions sur les types de retour. Elles doivent avoir un type de retour défini ou être capable de déduire le type de retour. Cela peut limiter leur flexibilité dans certains cas complexes.