

# Programmation multi- paradigmes en C++

Session #11 : Vers la programmation fonctionnelle en C++

Julien Deantoni, Thomas Soucheyre,  
Stephane Janel, Ken-Patrick Lehrmann,  
Ludovic Tessier, Leandro Fontoura  
Cupertino, Kevin Yeung

# Agenda

**01** Fonctions anonymes (lambda)

---

**02** Bibliothèque *algorithm*

---

**03** Bibliothèque *ranges* (C++20)

---

# 1.

## Fonctions anonymes (lambda)

# Utilisation des algorithmes de la STL

\_ La plupart des algorithmes de la STL ont besoin d'objets fonction (foncteurs)

Catégorie d'itérateur requise a minima

```
template< class ForwardIt, class UnaryPred >
ForwardIt remove_if( ForwardIt first, ForwardIt last, UnaryPred p );
```

```
template< class InputIt, class UnaryPred >
InputIt find_if( InputIt first, InputIt last, UnaryPred p );
```

# Utilisation des algorithmes de la STL

\_ La plupart des algorithmes de la STL ont besoin d'objets fonction (foncteurs)

Catégorie d'itérateur requise a minima

```
template< class ForwardIt, class UnaryPred >
ForwardIt remove_if( ForwardIt first, ForwardIt last, UnaryPred p );
```

```
template< class InputIt, class UnaryPred >
InputIt find_if( InputIt first, InputIt last, UnaryPred p );
```

Prédicat unaire = fonction avec un seul paramètre et qui retourne un booléen

# Utilisation des algorithmes de la STL

\_ La plupart des algorithmes de la STL ont besoin d'objets fonction (foncteurs)

Catégorie d'itérateur requise a minima

```
template< class ForwardIt, class UnaryPred >
ForwardIt remove_if( ForwardIt first, ForwardIt last, UnaryPred p );
```

```
template< class InputIt, class UnaryPred >
InputIt find_if( InputIt first, InputIt last, UnaryPred p );
```

Prédicat unaire = fonction avec un seul paramètre et qui retourne un booléen

```
template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

Fonction de comparaison avec 2 paramètres et qui retourne un booléen valant « true » si le premier paramètre est mal ordonné par rapport au second paramètre

# Exemple d'utilisation de *std::find\_if*

## \_Exemple simple

```
bool is_even(int i) {  
    return i % 2 == 0;  
}  
  
bool contains_even_numbers(const std::vector<int> &v) {  
    return std::any_of(v.begin(), v.end(), is_even);  
}
```

La fonction `is_even` respecte les conditions d'un prédicat unaire

# Exemple d'utilisation de *std::find\_if*

## \_Exemple simple

```
bool is_even(int i) {
    return i % 2 == 0;
}

bool contains_even_numbers(const std::vector<int> &v) {
    return std::any_of(v.begin(), v.end(), is_even);
}
```

La fonction `is_even` respecte les conditions d'un prédicat unaire

## \_Comment généraliser ?

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {
    return std::any_of(v.begin(), v.end(), ??);
}
```

Besoin d'une fonction avec 2 paramètres, donc ce n'est plus un prédicat unaire



# Foncteur

Foncteur = struct définissant l'opérateur () qui permet d'appeler l'instance comme une fonction

```
struct DivisibleBy {
    bool operator()(int i) const {
        return _n == 0 ? false : i % _n == 0;
    }
    int _n;
};
```

```
bool contains numbers divisible by_n(int n, const std::vector<int> &v) {
    DivisibleBy divisible_by_n{n};
    for (int i : v) {
        if (divisible_by_n(i)) {
            return true;
        }
    }
    return false;
}
```

Instantiation du foncteur

Appel de l'opérateur ()

# Foncteur

Foncteur = struct définissant l'opérateur () qui permet d'appeler l'instance comme une fonction

```
struct DivisibleBy {
    bool operator()(int i) const {
        return _n == 0 ? false : i % _n == 0;
    }
    int _n;
};
```

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {
    DivisibleBy divisible_by_n{n};
    for (int i : v) {
        if (divisible_by_n(i)) {
            return true;
        }
    }
    return false;
}
```

Instantiation du foncteur

Appel de l'opérateur ()

Equivalent avec `std::find_if`

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {
    return std::any_of(v.begin(), v.end(), DivisibleBy{n});
}
```

# Foncteur

Foncteur = struct définissant l'opérateur () qui permet d'appeler l'instance comme une fonction

```
struct DivisibleBy {
    bool operator()(int i) const {
        return _n == 0 ? false : i % _n == 0;
    }
    int _n;
};
```

Plutôt fastidieux à écrire, n'est-ce pas ?

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {
    DivisibleBy divisible_by_n{n};
    for (int i : v) {
        if (divisible_by_n(i)) {
            return true;
        }
    }
    return false;
}
```

Instanciation du foncteur

Appel de l'opérateur ()

Equivalent avec *std::find\_if*

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {
    return std::any_of(v.begin(), v.end(), DivisibleBy{n});
}
```

# Fonction anonyme

\_ Fonction anonyme / fonction lambda / lambda expression

- <https://en.cppreference.com/w/cpp/language/lambda>

\_ Syntaxe de base: `[captures] (paramètres) { corps de la fonction }`

# Fonction anonyme

\_ Fonction anonyme / fonction lambda / lambda expression

- <https://en.cppreference.com/w/cpp/language/lambda>

\_ Syntaxe de base: [captures] (paramètres) { corps de la fonction }

\_ Exemple

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {  
    return std::any_of(v.begin(), v.end(), [n](int i) -> bool {  
        return n == 0 ? false : i % n == 0;  
    });  
}
```

# Fonction anonyme

\_ Fonction anonyme / fonction lambda / lambda expression

- <https://en.cppreference.com/w/cpp/language/lambda>

\_ Syntaxe de base: [captures] (paramètres) { corps de la fonction }

\_ Exemple

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {
    return std::any_of(v.begin(), v.end(), [n](int i) -> bool {
        return n == 0 ? false : i % n == 0;
    });
}
```

Capture de la variable n par valeur (copie)

# Fonction anonyme

\_ Fonction anonyme / fonction lambda / lambda expression

- <https://en.cppreference.com/w/cpp/language/lambda>

\_ Syntaxe de base: [captures] (paramètres) { corps de la fonction }

\_ Exemple

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {
    return std::any_of(v.begin(), v.end(), [n](int i) -> bool {
        return n == 0 ? false : i % n == 0;
    });
}
```

Paramètre de la fonction

Capture de la variable n par  
valeur (copie)

# Fonction anonyme

\_ Fonction anonyme / fonction lambda / lambda expression

- <https://en.cppreference.com/w/cpp/language/lambda>

\_ Syntaxe de base: [captures] (paramètres) { corps de la fonction }

\_ Exemple

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {
    return std::any_of(v.begin(), v.end(), [n](int i) -> bool {
        return n == 0 ? false : i % n == 0;
    });
}
```

Paramètre de la fonction

Type de la valeur retournée  
(optionnel)

Capture de la variable n par  
valeur (copie)



# Fonction anonyme

\_ Fonction anonyme / fonction lambda / lambda expression

- <https://en.cppreference.com/w/cpp/language/lambda>

\_ Syntaxe de base: [captures] (paramètres) { corps de la fonction }

\_ Exemple

```
bool contains_numbers_divisible_by_n(int n, const std::vector<int> &v) {
    return std::any_of(v.begin(), v.end(), [n](int i) -> bool {
        return n == 0 ? false : i % n == 0;
    });
}
```

Paramètre de la fonction

Type de la valeur retournée  
(optionnel)

Capture de la variable n par  
valeur (copie)

\_ Possibilité d'assigner l'expression à une variable

```
auto divisible_by_n = [n](int i) { return n == 0 ? false : i % n == 0; };

std::function<bool(int)> f = [n](int i) { return n == 0 ? false : i % n == 0; };
```

# Fonction anonyme

## \_Captures

- [] aucune capture de variable extérieure
- [n, a] capture les variables n et a par copie
- [&n] capture la variable n par référence
- [=] capture toutes les variables par copie
- [&] capture toutes les variables par référence
- [this] capture l'instance de la classe courante et tous ses membres par référence
- [\*this] capture l'instance de la classe courante et tous ses membres par copie (depuis C++17)
- [=, &i] capture la variable i par référence et toutes les autres variables par valeur
  - Attention: « this » est capturé par référence
  - <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#f54-when-writing-a-lambda-that-captures-this-or-any-class-data-member-dont-use--default-capture>
- [&, i] capture la variable i par valeur et toutes les autres variables par référence



= et & sont appelés « capture-defaults » et doivent être situés en premier dans la liste

## \_Bonne pratique

- Lisibilité : pas plus de 4 lignes

# 2.

## Bibliothèque *algorithm*

# Bibliothèque *algorithm*

\_ Fournit des fonctions de base pour la plupart des algorithmes

- Recherche
- Tri
- Manipulation sur des objets itérables (copie, suppression, transformation, ...)

\_ A utiliser sans modération

- Aide à la compréhension du code (augmente l'abstraction)
- Evite les erreurs (par exemple, supprimer tous les nombres impairs d'un vecteur)
- Garantie une complexité algorithmique

\_ Seul inconvénient : écriture peu concise à cause des itérateurs

\_ Documentation complète

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

\_ Les utilisations les plus courantes suivent

|  |  |  |  |
|--|--|--|--|
| <b>Constrained algorithms and algorithms on ranges (C++20)</b>   |  |  |  |
| Constrained algorithms, e.g. <code>ranges::copy</code> , <code>ranges::sort</code> , ...   |  |  |  |
| <b>Execution policies (C++17)</b>  |  |  |  |
| <code>is_execution_policy (C++17)</code>   | <code>execution::seq (C++17)</code><br><code>execution::par (C++17)</code><br><code>execution::par_unseq (C++17)</code><br><code>execution::unseq (C++20)</code>   | <code>execution::sequenced_policy (C++17)</code><br><code>execution::parallel_policy (C++17)</code><br><code>execution::parallel_unsequenced_policy (C++17)</code><br><code>execution::parallel_unsequenced (C++20)</code> |  |
| <b>Non-modifying sequence operations</b>   |  | <b>Sorting and related operations</b>  |  |
| <b>Batch operations</b>  |  | <b>Partitioning operations</b>   |  |
| <code>for_each</code>  | <code>for_each_n (C++17)</code>  | <code>partition</code>   | <code>is_partitioned (C++11)</code>  |
| <b>Search operations</b>   |  | <code>partition_copy (C++11)</code>  | <code>partition_point (C++11)</code>   |
| <code>all_of (C++11)</code><br><code>any_of (C++11)</code><br><code>none_of (C++11)</code><br><code>count</code><br><code>count_if</code><br><code>mismatch</code><br><code>equal</code> | <code>find</code><br><code>find_if</code><br><code>find_if_not (C++11)</code><br><code>find_end</code><br><code>find_first_of</code><br><code>adjacent_find</code><br><code>search</code><br><code>search_n</code> | <code>stable_partition</code>  |  |
| <b>Modifying sequence operations</b>   |  | <b>Sorting operations</b>  |  |
| <b>Copy operations</b>   |  | <code>sort</code>  | <code>is_sorted (C++11)</code>   |
| <code>copy</code><br><code>copy_if (C++11)</code><br><code>copy_backward</code>  | <code>copy_n (C++11)</code><br><code>move (C++11)</code><br><code>move_backward (C++11)</code>   | <code>stable_sort</code>   | <code>is_sorted_until (C++11)</code>   |
| <b>Swap operations</b>   |  | <code>partial_sort</code>  | <code>nth_element</code>   |
| <code>swap</code><br><code>iter_swap</code>  | <code>swap_ranges</code>   | <code>partial_sort_copy</code>   |  |
| <b>Transformation operations</b>   |  | <b>Binary search operations (on partitioned ranges)</b>  |  |
| <code>replace</code><br><code>replace_if</code><br><code>transform</code>  | <code>replace_copy</code><br><code>replace_copy_if</code>  | <code>lower_bound</code><br><code>upper_bound</code>   | <code>equal_range</code><br><code>binary_search</code>   |
| <b>Generation operations</b>   |  | <b>Set operations (on sorted ranges)</b>   |  |
| <code>fill</code><br><code>fill_n</code>   | <code>generate</code><br><code>generate_n</code>   | <code>includes</code><br><code>set_union</code><br><code>set_intersection</code>   | <code>set_difference</code><br><code>set_symmetric_difference</code>   |
| <b>Removing operations</b>   |  | <b>Merge operations (on sorted ranges)</b>   |  |
| <code>remove</code><br><code>remove_if</code><br><code>unique</code>   | <code>remove_copy</code><br><code>remove_copy_if</code><br><code>unique_copy</code>  | <code>merge</code>   | <code>inplace_merge</code>   |
| <b>Order-changing operations</b>   |  | <b>Heap operations</b>   |  |
| <code>reverse</code><br><code>reverse_copy</code><br><code>rotate</code><br><code>rotate_copy</code>   | <code>random_shuffle (until C++17)</code><br><code>shuffle (C++11)</code><br><code>shift_left (C++20)</code><br><code>shift_right (C++20)</code>   | <code>push_heap</code><br><code>pop_heap</code><br><code>make_heap</code>  | <code>sort_heap</code><br><code>is_heap (C++11)</code><br><code>is_heap_until (C++11)</code>   |
| <b>Sampling operations</b>   |  | <b>Minimum/maximum operations</b>  |  |
| <code>sample (C++17)</code>  |  | <code>max</code><br><code>min</code><br><code>minmax (C++11)</code><br><code>clamp (C++17)</code>  | <code>max_element</code><br><code>min_element</code><br><code>minmax_element (C++11)</code>  |
| <b>Numeric operations</b>  |  | <b>Lexicographical comparison operations</b>   |  |
| <code>iota (C++11)</code><br><code>inner_product</code><br><code>adjacent_difference</code>  | <code>accumulate</code><br><code>reduce (C++17)</code><br><code>transform_reduce (C++17)</code>  | <code>lexicographical_compare</code><br><code>lexicographical_compare_three_way (C++20)</code>   |  |
| <b>Operations on uninitialized memory</b>  |  | <b>Permutation operations</b>  |  |
| <code>uninitialized_copy</code><br><code>uninitialized_move (C++17)</code><br><code>uninitialized_fill</code>  | <code>uninitialized_copy_n (C++11)</code><br><code>uninitialized_move_n (C++17)</code><br><code>uninitialized_fill_n</code>  | <code>next_permutation</code><br><code>prev_permutation</code>   | <code>is_permutation (C++11)</code>  |
|  |  | <b>C library</b>   |  |
|  |  | <code>qsort</code>   | <code>bsearch</code>   |
|  |  | <code>partial_sum</code><br><code>inclusive_scan (C++17)</code><br><code>exclusive_scan (C++17)</code>   | <code>transform_inclusive_scan (C++17)</code><br><code>transform_exclusive_scan (C++17)</code>   |
|  |  | <code>destroy (C++17)</code><br><code>destroy_n (C++17)</code><br><code>destroy_at (C++17)</code><br><code>construct_at (C++20)</code>   | <code>uninitialized_default_construct (C++17)</code><br><code>uninitialized_value_construct (C++17)</code><br><code>uninitialized_default_construct_n (C++17)</code><br><code>uninitialized_value_construct_n (C++17)</code> |

# Algorithmes de recherche

`std::find` / `std::find_if`

`_std::find` et `std::find_if` retournent un itérateur

```
bool contains(int value, const std::vector<int> &container) {
    for (int i : container) {
        if (i == value) {
            return true;
        }
    }
    return false;
}
```



```
bool contains(int value, const std::vector<IntKeyValue> &container) {
    return std::find(container.begin(), container.end(), value) != container.end();
}
```

# Algorithmes de recherche

std::all\_of / std::none\_of / std::any\_of

```
bool are_all_positive(const std::vector<int> &v) {
    for (int i : v) {
        if (i <= 0) {
            return false;
        }
    }
    return true;
}
```



```
bool are_all_positive = std::all_of(v.begin(), v.end(), [](int i) { return i > 0; });
bool are_all_positive = std::none_of(v.begin(), v.end(), [](int i) { return i <= 0; });
bool contains_negative = std::any_of(v.begin(), v.end(), [](int i) { return i <= 0; });
```

# Algorithmes de recherche

std::count / std::count\_if

```
int count_even_number(const std::vector<int> &v) {  
    int nb_even = 0;  
    for (int i : v) {  
        if (i % 2 == 0) {  
            ++nb_even;  
        }  
    }  
    return nb_even;  
}
```



Préférer

```
int nb_even = std::count_if(v.begin(), v.end(), [](int i) {  
    return i % 2 == 0;  
});
```

# Algorithmes de tri et opérations associées

## `_std::sort / std::stable_sort`

- Quand aucune fonction de comparaison n'est fournie, utilisation de `std::less`

```
using KVPair = std::pair<int, int>;
std::vector<KVPair> container{{5, 0}, {3, 1}, {3, 2}, {1, 3}};

const auto lower_key = [](const KVPair &lhs, const KVPair &rhs) { return lhs.first < rhs.first; };
std::sort(container.begin(), container.end(), lower_key);
```



# Algorithmes de tri et opérations associées

## `_std::sort` / `_std::stable_sort`

- Quand aucune fonction de comparaison n'est fournie, utilisation de `std::less`

```
using KVPair = std::pair<int, int>;
std::vector<KVPair> container{{5, 0}, {3, 1}, {3, 2}, {1, 3}};

const auto lower_key = [](const KVPair &lhs, const KVPair &rhs) { return lhs.first < rhs.first; };
std::sort(container.begin(), container.end(), lower_key);
```

## `_std::lower_bound` / `_std::upper_bound` / `_std::binary_search` (recherche dichotomique)

```
std::vector<int> v{1, 2, 3, 4};
```

```
std::lower_bound(v.begin(), v.end(), 2); // == v.begin() + 1
std::lower_bound(v.begin(), v.end(), 0); // == v.begin()
```

Recherche le premier élément x tel que  $x \geq 2$

```
std::upper_bound(v.begin(), v.end(), 2); // == v.begin() + 2
std::upper_bound(v.begin(), v.end(), 0); // == v.begin()
```

Recherche le premier élément x tel que  $x > 2$

```
std::binary_search(v.begin(), v.end(), 2); // == true
std::binary_search(v.begin(), v.end(), 0); // == false
```

# Algorithmes de manipulation de conteneurs

## Filtrer des éléments

`_std::copy_if` pour filtrer et copier vers un nouveau conteneur

```
std::vector<int> v{1, 4, 2, 3};  
std::vector<int> v_even;  
  
// copy even numbers  
std::copy_if(v.begin(), v.end(), std::back_inserter(v_even), [](int i) { return i % 2 == 0; }));
```

Construit un foncteur qui appelle la fonction membre `push_back`

# Algorithmes de manipulation de conteneurs

## Filtrer des éléments

`_std::copy_if` pour filtrer et copier vers un nouveau conteneur

```
std::vector<int> v{1, 4, 2, 3};
std::vector<int> v_even;

// copy even numbers
std::copy_if(v.begin(), v.end(), std::back_inserter(v_even), [](int i) { return i % 2 == 0; }));
```

Construit un foncteur qui appelle la fonction membre `push_back`

`_std::remove_if` + `erase` pour modifier directement le conteneur

```
std::vector<int> v{1, 4, 2, 3};

// remove odd numbers
v.erase(std::remove_if(v.begin(), v.end(), [](int i) { return i % 2 != 0; }), v.end());
```

`std::remove_if` place tous les éléments à supprimer à la fin du conteneur et retourne un itérateur vers le premier élément à supprimer

# Algorithmes de manipulation de conteneurs

## Transformer des éléments

```
std::vector<int> v{1, 4, 2, 3};  
std::vector<std::string> v_string;  
  
std::transform(v.begin(), v.end(), std::back_inserter(v_string), [](int i) {  
    return std::to_string(i);  
});
```

# Algorithmes de manipulation de conteneurs

Accumuler des éléments (sous-bibliothèque *numeric*)

`_std::accumulate`

```
std::vector<int> v{1, 4, 2, 3};
```

```
int sum = std::accumulate(v.begin(), v.end(), 0); // == 10
```

```
int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<>{}); // == 24
```

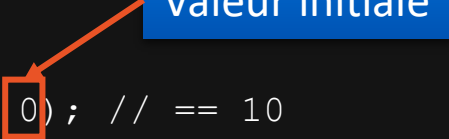
Valeur initiale

# Algorithmes de manipulation de conteneurs

Accumuler des éléments (sous-bibliothèque *numeric*)

## `_std::accumulate`

```
std::vector<int> v{1, 4, 2, 3};  
  
int sum = std::accumulate(v.begin(), v.end(), 0); // == 10  
  
int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<>{}); // == 24
```



## `_std::reduce`

- Equivalent à `std::accumulate` à l'exception que les éléments peuvent être groupés et réarrangés de manière arbitraire
- Le résultat est **non déterministe** si l'opération n'est pas associative ou pas commutative (comme l'addition des nombres flottants)

# Algorithmes de manipulation de conteneurs

`std::inner_product` (sous-bibliothèque *numeric*)

\_Utile pour le produit matriciel

```
std::vector<int> a{1, 4, 2, 3};  
std::vector<int> b{4, 1, 2, 0};  
  
int r = std::inner_product(a.begin(), a.end(), b.begin(), 0);  
// r = 1 * 4 + 4 * 1 + 2 * 2 + 3 * 0 = 12
```

Valeur initiale

\_`std::transform_reduce` est équivalent mais a les mêmes limitations que `std::reduce`

# Algorithmes de manipulation de conteneurs

Mélanger aléatoirement des éléments

```
std::vector<int> v(100);  
std::iota(v.begin(), v.end(), 0);
```

← Crée un vecteur rempli avec des entiers de 0 à 99

```
std::random_device rd;  
std::shuffle(v.begin(), v.end(), std::mt19937(rd()));
```



# 3.

## Bibliothèque *ranges* (C++20)

# Bibliothèque *ranges*

\_ Extension et généralisation de la bibliothèque *algorithm*

- Motivation première d'avoir une syntaxe plus concise

\_ Un « range » est un concept qui représente une séquence itérable, identifiée par un itérateur de début et une valeur spéciale de fin

\_ Une « view » est un objet

- Respectant le concept de « range »
- Qui se construit, copie, déplace, détruit en temps constant
- Exemple
  - Une paire d'itérateurs [begin, end)
  - Un itérateur et un prédicat de fin [begin, predicate)

\_ La plupart des algorithmes utilisant des paires d'itérateurs ont désormais un équivalent qui accepte un objet respectant le concept de « range »

\_ Les adaptateurs (range adaptors) sont des « views » qui peuvent être assemblées pour former des « pipelines » évaluées au fur et à mesure qu'on itère (**lazy evaluation**)

\_ Documentation complète

- <https://en.cppreference.com/w/cpp/ranges>

```
std::vector<int> v{1, 4, 2, 3};  
auto value_it = std::ranges::find(v, 2);
```

Conteneur utilisé directement en paramètre

```
bool are_all_positive = std::ranges::all_of(v, [](int i) { return i > 0; });
```

```
auto [rm_begin, rm_end] = std::ranges::remove_if(v, [](int i) { return i % 2 != 0; });  
v.erase(rm_begin, rm_end);
```

*remove\_if* retourne un « range »  
Préférer *std::erase\_if* pour un vecteur

```
std::vector<std::string> v_string;  
std::ranges::transform(v, std::back_inserter(v_string), [](int i) {  
    return std::to_string(i);  
});
```

```
std::optional<std::string> s = std::ranges::fold_left_first(v_string,  
    [](std::string a, const std::string &b) { return std::move(a) + '-' + b; }  
);
```

*std::ranges::fold\_left* remplace *std::accumulate* (C++23)

# Adaptateurs

Pas besoin de conteneur intermédiaire comme pour *copy\_if*

```
std::vector<int> v{1, 4, 2, 3};
for (int i : std::views::filter(v, [](int i) { return i % 2 == 0; })) {
    std::cout << i << std::endl;
}
```

Retourne « true » pour les éléments à garder

```
std::vector<int> v{1, 4, 2, 3};
for (int i : std::views::transform(v, [](int i) { return i * i; })) {
    std::cout << i << std::endl;
}
```

```
std::vector<int> v{1, 4, 2, 3};
for (const auto [index, num] : std::views::enumerate(v)) {
    std::cout << std::format("Number {} is at index {}\n", num, index);
}
```

C++23

```
std::vector<int> keys{0, 1, 2, 3};
std::vector<int> values{1, 4, 2, 3};
for (const auto [k, v] : std::views::zip(keys, values)) {
    std::cout << std::format("Key {} has value {}\n", k, v);
}
```

C++23

# Pipelines d'adaptateurs

```
auto even = [](int i) { return i % 2 == 0; };
auto square = [](int i) { return i * i; };

for (int i = 0; i < 6; ++i) {
    if (even(i)) {
        std::cout << square(i) << std::endl;
    }
}
```



Equivalent, décomposition en opérations simples

```
for (int i : std::views::iota(0, 6) | std::views::filter(even) | std::views::transform(square)) {
    std::cout << i << std::endl;
}
```

# Pipelines d'adaptateurs

Palindrome avec la bibliothèque *ranges*

```
bool palindrome(std::string_view s) {  
    const auto not_a_space = [](auto c) { return c != ' '; };  
    return std::ranges::equal(  
        s | std::views::filter(not_a_space),  
        s | std::views::filter(not_a_space) | std::views::reverse  
    );  
}
```