

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 );
```

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

```
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 );
```

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

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

```
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 prédictat 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}; // Instanciation du foncteur
    for (int i : v) {
        if (divisible_by_n(i)) // Appel de l'opérateur ()
            return true
    }
    return false;
}
```

Instanciation 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;
}
```

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});
```

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 renournée
(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

Constrained algorithms and algorithms on ranges (C++20)

Constrained algorithms, e.g. `ranges::copy`, `ranges::sort`, ...

Execution policies (C++17)

<code>is_execution_policy(C++17)</code>	<code>execution::seq (C++17)</code>	<code>execution::sequenced_policy (C++17)</code>
	<code>execution::par (C++17)</code>	<code>execution::parallel_policy (C++17)</code>
	<code>execution::par_unseq (C++17)</code>	<code>execution::parallel_unsequenced_policy (C++17)</code>
	<code>execution::unseq (C++20)</code>	<code>execution::parallel_unsequenced (C++20)</code>

Non-modifying sequence operations

Batch operations

`for_each` `for_each_n (C++17)`

Search operations

<code>all_of (C++11)</code>	<code>find</code>	<code>execution::sequenced_policy (C++17)</code>
<code>any_of (C++11)</code>	<code>find_if</code>	<code>execution::parallel_policy (C++17)</code>
<code>none_of (C++11)</code>	<code>find_if_not (C++11)</code>	<code>execution::parallel_unsequenced_policy (C++17)</code>
<code>count</code>	<code>find_end</code>	<code>execution::parallel_unsequenced (C++20)</code>
<code>count_if</code>	<code>find_first_of</code>	
<code>mismatch</code>	<code>adjacent_find</code>	
<code>equal</code>	<code>search</code>	
	<code>search_n</code>	

Modifying sequence operations

Copy operations

<code>copy</code>	<code>copy_n (C++11)</code>	<code>execution::sequenced_policy (C++17)</code>
<code>copy_if (C++11)</code>	<code>move (C++11)</code>	<code>execution::parallel_policy (C++17)</code>
<code>copy_backward</code>	<code>move_backward (C++11)</code>	<code>execution::parallel_unsequenced_policy (C++17)</code>

Swap operations

<code>swap</code>	<code>swap_ranges</code>
-------------------	--------------------------

Transformation operations

<code>replace</code>	<code>replace_copy</code>
<code>replace_if</code>	<code>replace_copy_if</code>
<code>transform</code>	

Generation operations

<code>fill</code>	<code>generate</code>
<code>fill_n</code>	<code>generate_n</code>

Removing operations

<code>remove</code>	<code>remove_copy</code>
<code>remove_if</code>	<code>remove_copy_if</code>
<code>unique</code>	<code>unique_copy</code>

Order-changing operations

<code>reverse</code>	<code>random_shuffle (until C++17)</code>
<code>reverse_copy</code>	<code>shuffle (C++11)</code>
<code>rotate</code>	<code>shift_left (C++20)</code>
<code>rotate_copy</code>	<code>shift_right (C++20)</code>

Sampling operations

<code>sample (C++17)</code>	
-----------------------------	--

Numeric operations

<code>iota (C++11)</code>	<code>accumulate</code>
<code>inner_product</code>	<code>reduce (C++17)</code>
<code>adjacent_difference</code>	<code>transform_reduce (C++17)</code>

Operations on uninitialized memory

<code>uninitialized_copy</code>	<code>uninitialized_copy_n (C++11)</code>
<code>uninitialized_move (C++17)</code>	<code>uninitialized_move_n (C++17)</code>
<code>uninitialized_fill</code>	<code>uninitialized_fill_n</code>

<code>execution::sequenced_policy (C++17)</code>	<code>execution::parallel_policy (C++17)</code>
<code>execution::parallel_unsequenced_policy (C++17)</code>	<code>execution::parallel_unsequenced (C++20)</code>

Sorting and related operations

<code>partition</code>	<code>is_partitioned (C++11)</code>
<code>partition_copy (C++11)</code>	<code>partition_point (C++11)</code>
<code>stable_partition</code>	

Sorting operations

<code>sort</code>	<code>is_sorted (C++11)</code>
<code>stable_sort</code>	<code>is_sorted_until (C++11)</code>
<code>partial_sort</code>	<code>nth_element</code>
<code>partial_sort_copy</code>	

Binary search operations (on partitioned ranges)

<code>lower_bound</code>	<code>equal_range</code>
<code>upper_bound</code>	<code>binary_search</code>

Set operations (on sorted ranges)

<code>includes</code>	<code>set_difference</code>
<code>set_union</code>	<code>set_symmetric_difference</code>
<code>set_intersection</code>	

Merge operations (on sorted ranges)

<code>merge</code>	<code>inplace_merge</code>
--------------------	----------------------------

Heap operations

<code>push_heap</code>	<code>sort_heap</code>
<code>pop_heap</code>	<code>is_heap (C++11)</code>
<code>make_heap</code>	<code>is_heap_until (C++11)</code>

Minimum/maximum operations

<code>max</code>	<code>max_element</code>
<code>min</code>	<code>min_element</code>
<code>minmax (C++11)</code>	<code>minmax_element (C++11)</code>
<code>clamp (C++17)</code>	

Lexicographical comparison operations

<code>lexicographical_compare</code>	
<code>lexicographical_compare_three_way (C++20)</code>	

Permutation operations

<code>next_permutation</code>	<code>is_permutation (C++11)</code>
<code>prev_permutation</code>	

C library

<code>qsort</code>	<code>bsearch</code>
<code>partial_sum</code>	<code>transform_inclusive_scan (C++17)</code>
<code>inclusive_scan (C++17)</code>	<code>transform_exclusive_scan (C++17)</code>
<code>exclusive_scan (C++17)</code>	

<code>destroy (C++17)</code>	<code>uninitialized_default_construct (C++17)</code>
<code>destroy_n (C++17)</code>	<code>uninitialized_value_construct (C++17)</code>
<code>destroy_at (C++17)</code>	<code>uninitialized_default_construct_n (C++17)</code>
<code>construct_at (C++20)</code>	<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;
}
```



Préférer

```
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;
}
```



```
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
```

Valeur initiale

std::reduce

- Équivalent 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 prédictat 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>

Algorithmes

Exemples

```
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

```
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;
}
```

Pas besoin de conteneur intermédiaire comme pour *copy_if*

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
    );
}
```