13

Formez les rang(e)s !

Jusqu'ici, vous avez découvert de nombreux algorithmes de la bibliothèque standard (`std::find`, `std::copy_if`, `std::transform`, etc.). Puissants, souples, efficaces... mais parfois un peu verbeux, voire un brin fatigants à lire ou à écrire. Prenons un exemple simple : vous souhaitez extraire tous les nombres divisibles par trois d'un tableau, et calculer le carré de chaque élément. Avec les algorithmes classiques, cela se traduit souvent par une série de boucles ou d'appels imbriqués, le tout accompagné de nombreux itérateurs, de conteneurs temporaires et d'un code qui n'a rien de franchement lisible.

Heureusement, parmi les améliorations apportées par C++20, une nouvelle approche a vu le jour. Directement inspirée de la programmation fonctionnelle – comme les lambdas que nous venons de découvrir –, cette approche permet d'enchaîner des opérations de manière fluide, claire, et expressive.

Ce chapitre vous propose de découvrir les *ranges* et les *views* : une manière moderne de transformer, filtrer et manipuler des collections sans s'encombrer de détails techniques.

13.1. Un exemple concret

Voici un exemple d'implémentation de l'énoncé donné en introduction, tel que nous savons le faire depuis que nous connaissons les conteneurs et les algorithmes. Le code fonctionne très bien, mais il impose de créer une copie, de jongler avec `std::remove_if` et `erase` pour supprimer les éléments non désirés, puis d'utiliser `std::for_each` pour transformer les valeurs. Avouons-le : c'est un peu lourd.

```
import std;

int main()
{
  const std::vector nombres { 8, 3, 4, 12, 5, 6, 9 };
  std::vector copie = nombres;

  // Étape 1 : ne garder que ceux divisibles par trois.
  auto it = std::remove_if(
    std::begin(copie),
    std::end(copie),
    [](int n) { return n % 3 != 0; }
  );
```

```
  copie.erase(it, std::end(copie));
  std::println("{}", copie);

  // Étape 2 : les mettre au carré.
  std::for_each(
    std::begin(copie),
    std::end(copie),
    [](int& n) { n *= n; }
  );

  std::sort(std::begin(copie), std::end(copie));

  // Affichage du résultat.
  std::println("{}", copie);

  return 0;
}
```

Maintenant, admirez la version équivalente en utilisant les possibilités offertes par le C++ moderne. Le code est plus court, plus lisible, et les opérations s'enchaînent avec fluidité et naturel.

Note › *Si vous utilisez les fichiers d'en-tête, il faudra inclure le fichier* `<ranges>`.

```
import std;

int main()
{
  const std::vector nombres { 8, 3, 4, 12, 5, 6, 9 };

  auto copie = nombres
    | std::views::filter([](int n) { return n % 3 == 0; })
    | std::views::transform([](int n) { return n * n; })
    | std::ranges::to<std::vector<int>>();

  std::ranges::sort(copie);

  // Affichage du résultat.
  std::println("{}", copie);

  return 0;
}
```

13.2. Qu'est-ce qu'un *range* et une *view* ?

Les *ranges* représentent une séquence de données, sur laquelle on peut itérer. Cela inclut donc les conteneurs, les chaînes de caractères et d'autres choses encore. Ce qui distingue un *range* des itérateurs classiques, c'est qu'un range nous permet de travailler avec les éléments sans avoir à nous soucier des détails de leur stockage ou de leur organisation interne. En d'autres termes, un *range* masque la gestion des itérateurs, et nous permet de nous concentrer sur ce que nous faisons avec les données.

Quant aux *views*, ce sont des *ranges* particuliers. Elles sont utilisées pour appliquer des opérations de transformation ou de filtrage sur une collection, *sans en modifier le contenu original*, ni faire de copie en mémoire. Elles agissent plutôt comme une projection ou un filtre des données sous-jacentes.

Ainsi, dans l'exemple précédent, `nombres` est un *range* sur lequel on applique deux *views*. La première filtre les éléments pour ne garder que ceux qui sont divisibles par trois. La deuxième transforme ensuite chaque élément pour obtenir son carré. Chaque opération s'enchaîne de manière fluide avec l'opérateur `|` qu'on appelle le *pipe*.

Note *› Les Linuxiens parmi vous connaissent probablement le pipe, puisque c'est de cette manière qu'on combine, en ligne de commande, la sortie d'un programme avec l'entrée d'un autre.*

13.3. Réécrivons le passé

Dans les chapitres sur les conteneurs et sur les lambdas, nous avons vu de nombreux exemples d'algorithmes utilisant les itérateurs. Que diriez-vous de les réécrire sous forme de *ranges* et de *views*, pour comparer ?

std::ranges::find — trouver un certain élément

Comme pour la version à base d'itérateurs, `std::ranges::find` cherche un élément dans un ensemble et renvoie un itérateur sur le premier trouvé. La version `std::ranges::find_if` attend en plus un prédicat unaire à respecter. Le code suivant trouve le premier `10` mais aussi le premier nombre pair rencontré.

```
import std;

int main()
{
  const std::vector nombres { 1, 2, 4, 10, 57, 84, 10 };

  const auto it_10 = std::ranges::find(nombres, 10);
```