# Découvrir les algorithmes standards

La page de documentation sur les algorithmes de la bibliothèque standard fournit une liste d'une centaine de fonction différentes. A cela, il faut ajouter le fait que chaque fonction peut avoir plusieurs signatures (listes de paramètres différentes, en particulier des versions avec et sans prédicat personnalisé).

On ne va pas détailler dans ce cours tout les algorithmes. Cela serait purement descriptif et vous n'apprendrez pas grand chose de plus qu'en lisant la documentation. Ce qui est important est d'avoir une vision d'ensemble des algorithmes proposés, savoir trouver l'algorithme qui nous intéresse lorsque l'on est face à une problématique.

Nous allons voir dans ce chapitre quelques algorithmes importants, pour introduire quelques notions importantes pour utiliser au mieux les algorithmes, et apprendre à lire la page de documentation. La fin du chapitre sera fortement orienté sur des exercices, pour vous aider à acquérir une pratique.

## Les itérateurs

Pour appeler les algorithmes sur des collections, vous avez vu les fonctions `begin` et `end` pour parcourir du début à la fin (et leur équivalent “reverse”, pour lire de la fin au début, `rbegin` et `rend`). Il est possible d'enregistrer ces positions dans des variables, par exemple en utilisant l'inférence de type avec `auto` :

```
std::vector<int> v { 1, 5, 2, 4, 3 };
auto b = begin(v);
auto e = end(v);
std::sort(b, e);
```

Bien sûr, il est également possible d'écrire explicitement le type des

variables `b` et `e`, au lieu d'utiliser l'inférence de type. La syntaxe est dans ce cas :

```
std::vector<int>::iterator b = begin(v);
std::vector<int>::iterator e = end(v);
```

Vous verrez par la suite la signification exacte de cette syntaxe, un peu compliquée (c'est la raison pour laquelle on utilise l'inférence de type, c'est plus simple à écrire). Ce qu'il est important de comprendre, c'est que la position dans une collection est gérée en C++ par le concept d'itérateur. Peut importe pour le moment de savoir à quoi correspond exactement les itérateurs, retenez qu'ils représente une position.

La paire d'itérateurs `b` et `e` définissent ce que l'on appelle un “range”. Plus généralement, un *range* est une paire d'itérateurs :

- qui proviennent de la même collection ;
- qui sont ordonnés (la première position est plus petite ou égale à la seconde).

Ainsi, les paires d'itérateurs `(begin(v), end(v))` ou `(end(v), end(v))` sont des *ranges*, tandis que les paires `(begin(v1), end(v2))` et `(end(v), begin(v))` ne le sont pas. Cette notion de *range* est importante, puisque les algorithmes de la bibliothèque standard n'acceptes en argument que des *range*.

Pour rappel, si ce n'est pas le cas, cela ne produit pas de message d'erreur, mais un comportement indéfini (*undefined behavior*), ce qui est une erreur très difficile à identifier.

## Créer une nouvelle collection

Il est possible de créer une nouvelle collection à partir d'un *range*, en passant une paire d'itérateurs en arguments (donc entre parenthèses) lors de la définition d'une variable.

```
vector<int> v1 { 1, 2, 3, 4, 5 };
```

```
// définition d'une nouvelle variable
vector<int> v2(begin(v1), end(v2));

// avec auto
auto v3 = vector<int>(begin(v1), end(v1));
```

Egalement possible de créer un nouvel objet et de l'affecter à une variable existante

```
vector<int> v4 {}; // vide
v4 = vector<int>(begin(v1), end(v1));
```

Bien sûr, en utilisant begin et end comme range, cela est peut intéressant, on fait l'équivalent d'une copie, donc on peut utiliser l'opérateur d'affection :

```
vector<int> v2 { v1 };
auto v3 = v1;
v4 = v1;
```

Contrairement à la copie, il est possible de changer le type de conteneur. Par exemple, pour passer à un vector<float>

```
vector<int> v1 { 1, 2, 3, 4, 5 };

vector<float> v2 { v1 };              // erreur
vector<float> v2(begin(v1), end(v1)); // ok
```

On va pouvoir également créer une sous-collection. Par exemple, pour créer une collection contenant la première moitié des éléments d'une collection :

```
vector<int> v1 { 1, 2, 3, 4, 5 };
vector<float> v2(begin(v1), end(v1) - v1.size() / 2);
```

ou les 2 premiers éléments :

```
vector<int> v1 { 1, 2, 3, 4, 5 };
vector<float> v2(begin(v1), begin(v1) + 2);
```

## Inclusion et exclusion des bornes dans un range

Pour rechercher dans une collection, on peut utiliser find. Si on regarde la documentation, indique que cette fonction retourne un itérateur (donc une position).

```
vector<int> v1 { 1, 2, 3, 4, 5 };
auto position = find(begin(v), end(v), 3);
```

On peut alors utiliser cette position pour créer deux sous-collection :

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> v1 { 1, 2, 3, 4, 5 };
    auto p = find(begin(v1), end(v1), 3);

    vector<float> v2(begin(v1), p);
    for (auto i: v2) cout << i << ' ';
    std::cout << std::endl;

    vector<float> v3(p, end(v1));
    for (auto i: v3) cout << i << ' ';
    std::cout << std::endl;
}
```

affiche :

```
1 2
3 4 5
```

On voit ici une particularité importante des *ranges*. La variable `p` correspond à la position de la valeur `3` dans la collection. On pourrait croire que la collection `v2`, qui est créée à partir de `begin(v1)` et `p`, allait contenir les éléments `{ 1, 2, 3 }`. Ce qui n'est pas le cas.

En effet, un *range* est une paire d'itérateurs dont le premier est inclut et le second exclu.

On écrire souvent `[first, last)` pour désigner un *range*. Cette notation mathématique permet d'écrire un ensemble, avec les crochets droits pour inclure les bornes et les parenthèses pour les exclure. Ainsi, l'ensemble [0, 1) correspond à l'ensemble des réels compris entre 0 inclut et 1 exclu, (0, 1) exclu les valeur 0 et 1, (0, 1] exclu 0 et inclut 1, etc.
Revenons sur un code précédent. Lors que l'on écrit :

```
vector<int> v1 { 1, 2, 3, 4, 5 };
vector<int> v2(begin(v1), end(v2));
```

On a bien une copie complète de `v1`. Est-ce que cela est compatible avec l'exclusion de la borne supérieure du *range* ? En fait, oui, tout simplement parce que end(v) correspond à la fin de la collection, pas au dernier élément de cette collection. Si on créés des sous-collection pour les premier et dernier éléments :

```
vector<int> v1 { 1, 2, 3, 4, 5 };

// premier élément ?
vector<int> v2(begin(v1), begin(v1));
cout << boolalpha << ve.empty() << endl; // vide

// premier élément
vector<int> v3(begin(v1), begin(v1)+1);
for (auto i: v2) cout << i << ' '; // ok, contient {1}

// dernier élément ?
vector<int> v4(end(v1), end(v1));
for (auto i: v4) cout << i << ' '; // vide

// dernier élément
vector<int> v5(end(v1)-1, end(v1));
for (auto i: v5) cout << i << ' '; // ok, contient {5}
```

En pratique, cela signifie que end(v) ne représente pas le dernier élément d'une collection, mais comme s'il y avait un élément supplémentaire après le dernier élément.

**faire une figure**

## Tester si une recherche échoue

Pourquoi ce façon de faire ? Avec find, si aucun élément trouvé, comme indiquer que la recherche échoue. Une solution serait que find retourne un booléeen en plus. Mais compliqué à gérer. Solution choisie, retourne end() si la recherche échoue.

Donc tester le résultat de find :

```
auto p = find(begin(v), end(v), 3);
std::cout << boolalpha << (p == end(v)) << endl;
p = find(begin(v), end(v), 10);
std::cout << boolalpha << (p == end(v)) << endl;
```

Comme end(v) ne représente pas un élément dans une collection, il est interdit de le manipuler comme les autres position. Ainsi, il est légal d'écrire begin(v)+1, puisque begin(v) est une position correspondante à un élément dans la collection (s'il y a au moins élément dans la collection), on a le droit de la manipuler. Au contraire, end(v)+1 est interdit. C'est pour cela que la notion begin(v)+n est dangereuse, puisque l'on peut écrire un position invalide, si n > size

advance ? next ?

## Exécuter plusieurs recherche

Supposons que l'on a plusieurs fois la valeur dans une collection. Comment les trouver toutes avec find ?

Recherche avec begin et end. Si on trouve un élément (donc si p != end), alors on refait la recherche entre p+1 et end pour trouver le 2ème

élément. Et ainsi de suite pour trouver tous les éléments.

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    std::string const s { "azertyazerty" };
    auto position = std::find(begin(s), end(s), 'e');
    auto const first = std::string(begin(s), position);
    auto const second = std::string(position, end(s));
    std::cout << first << std::endl;
    std::cout << second << std::endl;

    position = std::find(position+1, end(s), 'e');
    std::cout << std::string(begin(s), position) <<
std::endl;
    std::cout << std::string(position, end(s)) << std::endl;
}
```

affiche :

```
az
ertyazerty
azertyaz
erty
```

second find : recherche à partir du 'e' trouvé

remarque, pour collection vide ( {} ou ””), pas d'élément et begin == end

## transform

Rôle de cet algorithme : prendre une collection et appliquer une fonction f sur chaque élément. Cf la signature de la documentation. Contrairement aux algorithmes que l'on a déjà vu, il ne faut pas fournir un prédicat (ie un foncteur qui retourne un booléen), mais une fonction unaire ou binaire (selon la version de `transform` appelée).

La documentation précise la signature des fonctions :

```
Ret fun(const Type &a);                  // unaire
Ret fun(const Type1 &a, const Type2 &b); // binaire
```

Ce qui est important à comprendre est que les algorithmes peuvent avoir des signatures spécifiques. Il ne faut pas hésiter à consulter la documentation pour savoir la syntaxe exacte à utiliser.

Avec les lambda génériques, l'utilisation de transform est assez proche de ce que l'on a déjà utilisé. Version unaire :

```
transform<begin(v), end(v), begin(v), [](auto value){ return
value + 1; });
```

incrémente de 1 chaque élément

Version binaire :

```
transform<begin(v1), end(v1), begin(v2), begin(v1) [](auto
lhs, auto rhs){ return lhs + rhs; });
```

additionne deux vector

## Travaux dirigés

Page de doc : plusieurs catégories d'algorithmes :

- modifiant : l'algo modifie la collection que l'on passe en argument (sort, transform, etc.)
- non modifiant : l'algo ne modifie pas la collection (equal, find, etc)
- partitionning : sépare une collection en sous collection (tous les éléments sont retouvé dans les sous collection, sans être en double)
- sorting et recherche binaire : tris
- set, heap,
- minmax, numérique

- liste d'exos...
- ascii art : convertir une image en ASCI ART

| Chapitre précédent | Sommaire principal | Chapitre suivant |
| --- | --- | --- |