Découvrir les algorithmes standards

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

On ne va pas détailler dans ce cours tous 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 où trouver l'algorithme qui nous intéresse lorsque l'on est face à une problématique.

Nous allons voir dans ce chapitre quelques algorithmes notables, pour introduire quelques notions importantes, utiliser au mieux les algorithmes, et apprendre à lire la page de documentation. La fin du chapitre sera fortement orientée sur des exercices, pour vous aider à acquérir par la 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 = std::begin(v);
auto e = std::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 = std::begin(v);
std::vector<int>::iterator e = std::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 qui est important de comprendre, c'est que la position dans une collection est gérée en C++ par le concept d'itérateur. Peu importe pour le moment de savoir à quoi correspondent exactement les itérateurs, retenez qu'ils représentent une position et qu'ils permettent de parcourir les éléments d'un conteneur.

La paire d'itérateurs `b` et `e` définit ce que l'on appelle un “range”, un intervalle. 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'acceptent en argument que des *range*s.

Attention, si vous n'utilisez pas un *range* valide, 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 argument (donc entre parenthèses) lors de la définition d'une variable.

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

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

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

Il est également possible de créer un nouvel objet et de l'affecter à une variable existante.

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

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

```
std::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>`

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

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

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

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

ou les 2 premiers éléments :

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

```
std::vector<float> v2(std::begin(v1), std::begin(v1) + 2);
```

Inclusion et exclusion des bornes dans un range

Pour effectuer une recherche dans une collection, on peut utiliser `find`. Si on regarde la documentation, elle indique que cette fonction prend un *range* et un élément à chercher, elle retourne l'itérateur (donc une position) de l'élément dans la collection s'il a été trouvé, sinon l'itérateur `end`.

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

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

main.cpp

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

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

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

    std::vector<float> v3(p, std::end(v1));
    for (auto i: v3) std::cout << i << ' ';
    std::cout << 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 }`, mais ce n'est pas le cas.

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

On écrira 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 inclus et 1 exclu, (0, 1) exclu les valeurs 0 et 1, (0, 1] exclu 0 et inclus 1, etc.

Revenons sur un code précédent. Lorsque l'on écrit :

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

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, et non au dernier élément de cette collection. Si on crée des sous-collections depuis les premiers et derniers éléments :

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

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

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

// dernier élément ?
std::vector<int> v4(std::end(v1), std::end(v1));
std::cout << boolalpha << v4.empty() << std::endl; // vide
```

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

En pratique, cela signifie que `end(v)` ne représente pas le dernier élément d'une collection, mais un élément supplémentaire et imaginaire qui se trouverait après le dernier élément.

faire une figure

Tester si une recherche échoue

Pourquoi cette façon de faire ? Avec `find`, si aucun élément n'est trouvé, comment indiquer que la recherche a échoué ? Une solution serait que `find` retourne un booléen en plus. Mais ce serait compliqué à gérer. La solution choisie est qu'elle retourne `end()` si la recherche échoue.

Donc tester le résultat de `find` :

```
auto p = std::find(begin(v), end(v), 3);
std::cout << boolalpha << (p == end(v)) << endl;
p = std::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 positions. 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 un élément dans la collection), on a le droit de le manipuler. Au contraire, `end(v)+1` est interdit. C'est pour cela que la notion `begin(v)+n` est dangereuse, puisque l'on peut accéder à une position invalide, si `n > size()`.

advance ? next ?

Exécuter plusieurs recherches

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

On recherche d'abord sur l'intervalle qui nous intéresse avec `begin` et `end`. Puis, si on trouve un élément (donc si `p != end`), alors on peut refaire la recherche entre `p+1` et `end` pour trouver un deuxième élément. Et ainsi de suite, jusqu'à trouver tous les éléments.

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

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

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

affiche :

```
az
ertyazerty
azertyaz
erty
```

second `find` : recherche à partir du 'e' trouvé

Pour une collection vide ({} ou ""), il n'y a pas d'élément et `begin == end`.

transform

Rôle de cet algorithme : prendre une collection et appliquer une fonction `fun` 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 (i.e. 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 :

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

incrémente de 1 chaque élément.

Version binaire :

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

additionne deux à deux, les éléments des deux vecteurs.

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 collections (tous les éléments sont retrouvés dans les sous-collections, sans qu'il y ait de doublon)
- 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

Cours, C++