Les catégories d'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).

Ce cours ne va pas détailler tous ces 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 vous intéresse lorsque vous êtes face à une problématique.

Pour organiser un peu les algorithmes de la bibliothèque standard, la page de documentation sur cppreference.com sépare les algorithmes en plusieurs catégories. Vous allez voir dans ce chapitre quelques algorithmes notables, ce qui va permettre d'introduire quelques notions importantes, vous apprendre à utiliser au mieux les algorithmes et à lire la page de documentation.

Note : des exercices seront ajoutés par la suite à ce cours. Ils vous permettront d'étudier et pratiquer plus en détail ces algorithmes.

Les algorithmes non modifiants

Les algorithmes non modifiants (*non-modifying sequence operations*). Ce sont des algorithmes qui ne modifient pas les collections sur les quelles ils sont utilisés. Vous allez trouvé dans cette catégorie par exemple l'algorithme d'égalité `std::equal` que vous avez déjà vu, ainsi que les algorithmes de recherche (en premier lieu `std::find`), les algorithmes de comptage (`std::count`) et l'algorithme `std::for_each` (qui permet

d'appeler une fonction sur chaque élément d'une collection).

Un algorithme non modifiants va donc prendre une collection sans la modifier et retourner un résultat. La valeur retournée peut être utiliser directement ou enregistrée dans une variable, comme n'importe quelle valeur. Le type de la valeur retournée dépend de l'algorithme et du type de collection, le plus simple est d'utiliser l'inférence de type pour créer une variable (consulter la documentation pour connaitre le type exact).

Les exemples de code suivants utilisent une chaîne, pour simplifier l'écriture du code et l'affichage du résultat. N'oubliez pas qu'une chaîne est une collection de caractères.

std::all_of, std::any_of et std::none_of

Les algorithmes std::all_of, std::any_of et std::none_of permettent de tester respectivement : si tous les éléments d'une collection respectent un prédicat, si au moins un élément respecte un prédicat ou si aucun élément ne respecte un prédicat. Ces trois algorithmes retournent un booléen.

Par exemple, pour tester si une chaîne possède des majuscules, vous pouvez utiliser les fonctions définies dans l'en-tête <cctype>, en particulier la fonction `isupper` ("est une majuscule").

main.cpp

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

int main() {
    const std::string s { "abcDEFf" };
    std::cout << std::boolalpha;
    std::cout << std::all_of(begin(s), end(s), isupper) <<
std::endl;
    std::cout << std::any_of(begin(s), end(s), isupper) <<
```

```
std::endl;
    std::cout << std::none_of(begin(s), end(s), isupper) <<
std::endl;
}
```

affiche :

```
false
true
false
```

std::all_of, std::any_of et std::none_of

Les algorithmes std::count et std::count_if permettent le nombre d'éléments d'une collection correspondant respectivement a une valeur et un prédicat. Ces algorithmes retournent une valeur entière signée.

main.cpp

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

int main() {
    const std::string s {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::cout << std::count(begin(s), end(s), '7') << std::
endl;
    std::cout << std::count_if(begin(s), end(s), isdigit) <<
std::endl;
}
```

affiche :

```
3
21
```

std::equal

Et bien sur, vous avez déjà vu l'algorithme std::equal.

main.cpp

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

int main() {
    const std::string s1 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    const std::string s2 {
"8cc52221d9bd6a3701c90969bcee91be4810c8d5" };
    std::cout << std::boolalpha;
    std::cout << std::equal(begin(s1), end(s1), begin(s2),
end(s2)) << std::endl;
}
```

affiche :

```
false
```

Les algorithmes modifiants

Les algorithmes modifiants (*modifying sequence operations*), vous l'avez surement deviné, modifient les collections sur les quelles ils sont utilisés. Vous trouverez dans cette catégorie les algorithmes pour ajouter (`std::fill`), supprimer (`std::remove`), remplacer (`std::replace`), copier (`std::copy`), échanger (`std::swap`) ou mélanger (`std::suffle`) des éléments.

Les algorithmes modifiants sont de deux types : les algorithmes qui modifient directement une collection et les algorithmes qui utilisent une collection et modifient une autre collection.

main.cpp

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

```cpp
#include <algorithm>

int main() {
    std::string s { "1234567890" };
    std::fill(begin(s), end(s), 'A');
    std::cout << s << std::endl;
    std::fill_n(begin(s), 5, '1');
    std::cout << s << std::endl;
}
```

affiche :

```
AAAAAAAAAA
11111AAAAA
```

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 :

```cpp
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 :

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

incrémente de 1 chaque élément.

Version binaire :

```cpp
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.

Les algorithmes de partitionnement

Les algorithmes de partitionnement (*partitioning operations*) permettent séparer une collection en deux sous-collections.

Les algorithmes de tri

Les algorithmes de tri (*sorting operations*) permettent de trier les éléments d'une collection.

Les algorithmes binaires de recherche

Les algorithmes binaires de recherche (*binary search operations*) permettent de rechercher un élément dans une collection triée.

Les algorithmes sur les ensembles

Les algorithmes sur les ensembles (*set operations*) permettent de manipuler une collection comme un ensemble d'éléments (donc sans élément en double).

Les algorithmes sur les Tas

Les algorithmes sur les Tas (*heap operations*) permettent de manipuler une collection comme un Tas d'éléments.

Les algorithmes minimum/maximum

Les **algorithmes minimum/maximum** permettent de recherche des éléments minimum ou maximum et réaliser des permutations.

Les algorithmes numériques

Les **algorithmes numériques** (*numeric operations*) permettent de travailler sur des collections de nombres.

Chapitre précédent	Sommaire principal	Chapitre suivant