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.

Ce chapitre détaille plus particulièrement les algorithmes qui ne retournent pas d'itérateur et qui ne nécessitent donc pas de détailler ce concept. Les itérateurs seront détaillés dans le chapitre suivant, ainsi que d'autres algorithmes de la bibliothèques standard.

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 trouver 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 utilisée 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::count et std::count_if

Les algorithmes std::count et std::count_if retournent le nombre d'éléments d'une collection correspondant respectivement à 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 sûr, vous avez déjà vu l'algorithme std::equal.

main.cpp

```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.

Par exemple :

```
std::fill(begin(in), end(in), value);
// seulement in
std::transform(begin(in), end(in), begin(out), operation);
// in et out
```

Mais il est généralement possible d'utiliser le second type d'algo modifiant avec la même collection en entrée et sortie :

```
std::transform(begin(in), end(in), begin(in), operation);
```

Dans le premier cas, transforme les éléments de `in` et met le résultat dans `out`. Dans le second cas, transforme les éléments de `in` et met le résultat dans `in`. (les valeurs initiales sont donc perdues).

std::copy, std::copy_if et std::copy_n

Note : pas de contrôle de dépassement de collection.

Equivalent : `std::move`, mais sera vu plus tard.

main.cpp

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

int main() {
    const std::string s1 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::string s2       {
"....................................." };
    std::string s3       {
"....................................." };
    std::string s4       {
"....................................." };
```

```
    std::copy(cbegin(s1), cend(s1), begin(s2));
    std::cout << s2 << std::endl;

    std::copy_if(cbegin(s1), cend(s1), begin(s3), isalpha);
    std::cout << s3 << std::endl;

    std::copy_n(cbegin(s1), 10, begin(s4));
    std::cout << s4 << std::endl;
}
```

affiche :

```
f9c02b6c9da8943feaea4966ba7417d65de2fe7e
fcbcdafeaeabaddefee.....................
f9c02b6c9d.............................
```

std::copy_backward

Note : pas de contrôle de dépassement de collection.

Equivalent : `std::move_backward`, mais sera vu plus tard.

main.cpp

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

int main() {
    const std::string s1 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::string s2       {
"....................................." };
    std::string s3       {
"....................................." };

    std::copy(begin(s1) + 10, begin(s1) + 15, begin(s2) + 10);
    std::cout << s2 << std::endl;
```

```
    std::copy_backward(begin(s1) + 10, begin(s1) + 15, begin
(s3) + 10);
    std::cout << s3 << std::endl;
}
```

affiche :

```
..........a8943.......................
.....a8943............................
```

std::fill et std::fill_n

main.cpp

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

std::transform

Différent des autres algos, ne prend pas un prédicat (objet appelable qui retourne un booléen), mais prend en paramètre un opérateur unaire ou binaire (selon la version de `transform` appelée).

La documentation précise la signature des fonctions (in = input = entrée, out = output = sortie) :

```
// unaire
TypeOut fun1(const Type &a);
std::transform(std::begin(in), std::end(in), std::begin(out),
fun1);

// binaire
TypeOut fun2(const Type1 &a, const Type2 &b);
std::transform(std::begin(in1), std::end(in1), std::begin(
in2), std::begin(out), fun2);
```

Par exemple :

Version unaire :

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

int main() {
    std::string s { "azerty" };
    std::transform(begin(s), end(s), begin(s), toupper);
    std::cout << s << std::endl;
}
```

affiche :

```
AZERTY
```

Version binaire :

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

int main() {
    std::vector<int> v1 { 1, 2, 3, 4 };
    const std::vector<int> v2 { -1, 2, -1, 2 };
```

```
    std::transform(begin(v1), end(v1), begin(v2), begin(v1),
std::multiplies<int>());
    for (auto i: v1) std::cout << i << ' ';
    std::cout << std::endl;
}
```

affiche :

```
-1 4 -3 8
```

std::generate et std::generate_n

avec rand ?

std::remove et std::replace

main.cpp

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

int main() {
    std::string s1 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::remove(begin(s1), end(s1), '7');
    std::cout << s1 << std::endl;

    std::string s2 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::remove_if(begin(s2), end(s2), isdigit);
    std::cout << s2 << std::endl;

    const std::string s3 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::string s4       {
"......................................." };
```

```
    std::remove_copy(begin(s3), end(s3), begin(s4), '7');
    std::cout << s4 << std::endl;

    const std::string s5 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::string s6       {
"......................................" };
    std::remove_copy_if(begin(s5), end(s5), begin(s6),
isdigit);
    std::cout << s6<< std::endl;
}
```

affiche :

```
f9c02b6c9da8943feaea4966ba41d65de2feee7e
fcbcdafeaeabaddefeea4966ba7417d65de2fe7e
f9c02b6c9da8943feaea4966ba41d65de2fee...
fcbcdafeaeabaddefee..................
```

main.cpp

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

int main() {
    std::string s1 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::replace(begin(s1), end(s1), '7', '.');
    std::cout << s1 << std::endl;

    std::string s2 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::replace_if(begin(s2), end(s2), isdigit, '.');
    std::cout << s2 << std::endl;

    const std::string s3 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::string s4       {
"......................................" };
    std::replace_copy(begin(s3), end(s3), begin(s4), '7',
```

```
'.');
    std::cout << s4 << std::endl;

    const std::string s5 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::string s6       {
"....................................." };
    std::replace_copy_if(begin(s5), end(s5), begin(s6),
isdigit, '.');
    std::cout << s6<< std::endl;
}
```

affiche :

```
f9c02b6c9da8943feaea4966ba.41.d65de2fe.e
f.c..b.c.da....feaea....ba....d..de.fe.e
f9c02b6c9da8943feaea4966ba.41.d65de2fe.e
f.c..b.c.da....feaea....ba....d..de.fe.e
```

std::swap etstd::swap_ranges

std::swap ne travaille pas sur des collections en particulier.

main.cpp

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

int main() {
    std::string s1 { "azerty" };
    std::string s2 { "123456" };
    std::swap(s1, s2);
    std::cout << s1 << std::endl;
    std::cout << s2 << std::endl;

    int x { 123 };
    int y { 456 };
    std::swap(x, y);
    std::cout << x << std::endl;
```

```
    std::cout << y << std::endl;
}
```

affiche :

```
123456
azerty
456
123
```

main.cpp

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

int main() {
    std::string s1 { "azerty" };
    std::string s2 { "123456" };
    std::swap_ranges(begin(s1), begin(s1) + 3, begin(s2));
    std::cout << s1 << std::endl;
    std::cout << s2 << std::endl;
}
```

affiche :

```
123rty
aze456
```

iter_swap ?

reverse et reverse_copy

main.cpp

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

int main() {
    std::string s1 { "azerty" };
```

```
    std::reverse(begin(s1), end(s1));
    std::cout << s1 << std::endl;

    const std::string s2 { "azerty" };
    std::string s3       { "......" };
    std::reverse_copy(begin(s2), end(s2), begin(s3));
    std::cout << s3 << std::endl;
}
```

affiche :

```
ytreza
ytreza
```

rotate

main.cpp

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

int main() {
    std::string s1 { "azerty" };
    std::rotate(begin(s1), begin(s1) + 2, end(s1));
    std::cout << s1 << std::endl;

    const std::string s2 { "azerty" };
    std::string s3       { "......" };
    std::rotate_copy(begin(s2), begin(s2) + 2, end(s2),
begin(s3));
    std::cout << s3 << std::endl;
}
```

affiche :

```
ertyaz
ertyaz
```

shuffle

main.cpp

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

int main() {
    std::string s { "azerty" };
    std::random_device rd;
    std::mt19937 g(rd());
    std::shuffle(begin(s), end(s), g);
    std::cout << s << std::endl;
}
```

affiche par exemple :

```
yazert
```

unique

main.cpp

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

int main() {
    std::string s1 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::sort(begin(s1), end(s1));
    auto last = std::unique(begin(s1), end(s1));
    std::cout << std::string(begin(s1), last) << std::endl;
}
```

affiche :

```
0123456789abcdef
```

Les algorithmes de partitionnement

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

main.cpp

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

int main() {
    std::string s1 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::partition(begin(s1), end(s1), isdigit);
    std::cout << s1 << std::endl;
}
```

affiche :

```
792025669718943476694aeabaefaddcbdecfefe
```

Les algorithmes de tri

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

main.cpp

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

int main() {
    std::string s1 {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::sort(begin(s1), end(s1));
    std::cout << s1 << std::endl;
}
```

affiche :

```
012234445666677789999aaaabbccdddeeeeefff
```

Les algorithmes binaires de recherche

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

main.cpp

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

int main() {
    std::string s {
"f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::sort(begin(s), end(s));
    std::cout << s << std::endl;

    const auto result = binary_search(begin(s), end(s), 'a');
    std::cout << std::boolalpha << result << std::endl;
}
```

affiche :

```
012234445666677789999aaaabbccdddeeeeefff
true
```

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.

main.cpp

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

int main() {
    std::vector<int> v1(5);
    std::iota(begin(v1), end(v1), 0);
    for(const auto i: v1) { std::cout << i << ' '; } std::
cout << std::endl;

    std::cout << std::accumulate(begin(v1), end(v1), 0) <<
std::endl;

    const std::vector<int> v2 = { 2, 3, 4, 0, 1 };
    std::cout << std::inner_product(begin(v1), end(v1),
begin(v2), 0) << std::endl;
    // = 0 * 2 + 1 * 3 + 2 * 4 + 3 * 0 + 4 * 1

    std::vector<int> v3(v2.size());
    std::adjacent_difference(begin(v1), end(v1), begin(v3));
    for(const auto i: v3) { std::cout << i << ' '; } std::
```

```
cout << std::endl;

    std::vector<int> v4 { v2 };
    std::adjacent_difference(begin(v4), end(v4), begin(v4));
    for(const auto i: v4) { std::cout << i << ' '; } std::
cout << std::endl;
}
```

affiche :

```
0 1 2 3 4
10
15
0 1 1 1 1
2 1 1 -4 1
```


Chapitre précédent	Sommaire principal	Chapitre suivant