Introduction aux algorithmes standards

Même si les collections offrent plus de fonctionnalités que les concepts de base `begin` et `end`, il est déjà possible d'utiliser plusieurs types d'algorithmes de la bibliothèque standard. Les différents concepts plus avancés, tels que les itérateurs, vont être introduit en même temps que des exemples d'algorithmes (recherche d'un élément, trier les éléments d'une collection, etc). Et pour commencer, ce chapitre détaille les algorithmes de comparaison.

L'opérateurs d'égalité

Dans les chapitres précédents, vous avez vu l'utilisation de l'opérateur de comparaison `==` (*egal-to operator*). Les collections de la bibliothèque standard, comme `std::vector`, `std::array` ou `std::string`, fournissent également cet opérateur. Vous pouvez donc écrire :

main.cpp

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

int main() {
    std::string const s1 { "salut" };
    std::string const s2 { "salut" };
    std::string const s3 { "hello" };
    std::cout << std::boolalpha;
    std::cout << (s1 == s2) << std::endl;
    std::cout << (s1 == s3) << std::endl;
}
```

affiche :

```
true
```

```
false
```

Il ne faut pas oublier les parenthèses autour du test d'égalité pour `std::string`. L'opérateur `«` ayant un sens pour cette classe, le compilateur ne pourra pas savoir si vous souhaitez écrire :

```
std::cout << (s1 == s2) << std::endl;
std::cout << s1 == (s2 << std::endl);
```

De la même façon pour les tableaux, il est possible d'utiliser l'opérateur `==`.

main.cpp

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

int main() {
    std::vector<int> const v1 { 1, 2, 3, 4 };
    std::vector<int> const v2 { 1, 2, 3, 4 };
    std::vector<int> const v3 { 4, 3, 2, 1 };
    std::cout << std::boolalpha;
    std::cout << (v1 == v2) << std::endl;
    std::cout << (v1 == v3) << std::endl;
}
```

affiche :

```
true
false
```

Avec les collections, l'opérateur d'égalité compare les éléments de la collection un par un. S'il trouve une différence, il retourne `false`. S'il arrive à la fin de la collection sans trouver de différence, il retourne `true`.

Pour vérifier si deux chaînes sont différentes, il est possible d'utiliser l'opérateur booléen de négation `!` : `!(s1 == s2)`. Plus simplement, il est possible d'utiliser l'opérateur de comparaison `!=` (*not-equal-to operator*).

main.cpp

```
#include <iostream>
```

```cpp
#include <string>

int main() {
    std::string const s1 { "salut" };
    std::string const s2 { "salut" };
    std::string const s3 { "hello" };
    std::cout << std::boolalpha << (s1 != s2) << std::endl;
    std::cout << (s1 != s3) << std::endl;
}
```

affiche :

```
false
true
```

On voit ici que les opérateurs `==` et `!=` sont étroitement liés. Lorsque l'un de ces deux opérateurs est utilisable, il est habituel de pouvoir utiliser aussi l'autre opérateur.

Une classe qui propose l'opérateur d'égalité `==` respecte le concept de “comparable par égalité” (EqualityComparable). Ce concept précise que l'opérateur d'égalité doit suivre les propriétés suivantes :

- réflexivité : quelque soit `a`, `a == a` est toujours vrai ;
- commutativité : si `a == b`, alors `b == a` ;
- transitivité : si `a == b` et `b == c`, alors `a == c`.

Ce concept est assez classique, vous le retrouvez en mathématique dans la théorie des ensembles. vous voyez ici un point important : lorsqu'une classe définit un opérateur `==`, vous pouvez vous attendre à ce qu'elle suive un certain nombre de règles : elle suit une **sémantique**.

Du point de vue de l'utilisateur de cette classe, il pourra l'utiliser de la même façon qu'il utilise n'importe quelle classe respectant cette sémantique. Du point de vue du concepteur de la classe (ce que vous apprendrez à faire dans la suite de ce cours), il suffit de définir les sémantiques que vous souhaitez donner à votre classe, l'écriture de la classe en découlera.

Au contraire, le non respect d'une sémantique sera très perturbant pour

l'utilisateur - et une source d'erreur sans fin. Imaginez que l'opérateur `==` ne réalise pas un test d'égalité, mais permet de faire la concaténation de deux chaînes ? Ou n'importe quoi d'autres, selon la classe ? La cohérence et l'homogénéité des syntaxes sont des notions importantes pour faciliter la lecture d'un code (et donc éviter les erreurs).

Bien sûr, ces considérations s'appliquent à l'ensemble des sémantiques usuelles, en particulier celle que vous connaissez en mathématique (addition avec `+`, soustraction avec `-`, etc.)

Comparer les éléments un par un

Si vous souhaitez comparer les éléments de deux collections un par un, deux sous-ensembles de collections ou si vous souhaitez utiliser un prédicat différent, il ne sera pas possible d'utiliser l'opérateur d'égalité `==`. Dans ce cas, la bibliothèque standard fournit l'algorithme `std::equal` pour comparer si les éléments de deux collections.

Lorsque vous découvrez une nouvelle fonctionnalité, la première chose à faire est de regarder la documentation : std::equal. Cette page peut vous apprendre plusieurs choses :

- cet algorithme est défini dans le fichier d'en-tête `<algorithm>`, il faudra donc l'inclure dans votre code pour utiliser `std::equal` ;
- il existe quatre versions de cet algorithme :
  - le premier prend en argument le début et la fin d'une première collection et le début d'une seconde collection ;
  - le second est similaire au premier, avec un prédicat personnalisé ;
  - la troisième prend en argument le début et la fin de la première collection, puis de la seconde ;
  - le quatrième est similaire au troisième, avec un prédicat personnalisé.
- le prédicat doit respecter la signature suivante : `bool pred(const Type1 &a, const Type2 &b);`. C'est donc une fonction binaire - une fonction qui prend deux arguments - et retourne un booléen.

La page de documentation donne également des codes d'exemple d'utilisation de ces fonctions.

La première et deuxième version de `std::equal` doit être utilisé avec précaution. Ces deux fonctions comparent les éléments un par un et s'arrête à la fin de la première collection. Si la seconde collection est plus grande que la première (par exemple les chaînes `"abcd"` et `"abcdEF"`), la comparaison se terminera sans prendre en compte les derniers éléments et `std::equal` retournera vrai, alors que ce n'est pas forcément le cas.

Si la seconde collection est plus petite que la première, `std::equal` continuera de travailler après la fin de la seconde collection, ce qui produira un crash, voire un comportement indéfini.

Dans les deux cas, il est possible de résoudre le problème en comparant les tailles respectives de deux collections avant d'utiliser `std::equal`. Si les tailles sont différentes, alors il n'est pas nécessaire d'utiliser `std::equal` dans ce cas.

La troisième (et quatrième) version de `std::equal` permet de tester si l'algorithme est arrivé à la fin des deux collections et donc qu'elle sont parfaitement identiques. Voici un code d'exemple pour illustrer ce problème :

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

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

```
    std::cout << std::equal(begin(s2), end(s2), begin(s1),
end(s1)) << std::endl;
}
```

affiche :

```
true
false

false
false
```

Par défaut, préférez l'utilisation de la version prenant le début et la fin des deux collections (les versions 3 et 4 de `std::equal`).

Attention aux éléments que vous passez en argument dans les fonctions. Le compilateur vérifie que vous passez des collections de types identiques en argument, pas que les informations passées. Par exemple, si vous passez en argument un élément d'une collection puis un élément d'une seconde collection, cela n'a pas de sens de parcourir une collection entre ces deux éléments.

```
std::equal(begin(s1), end(s2), begin(s2)); // problème, les
deux premiers arguments
                                           // ne proviennent
pas de la même collection
```

Dans ce cas, ce code ne produit pas d'erreur de compilation, mais un comportement indéterminé (*undefined behavior*, UB). Ce type d'erreur est assez complexe à détecter et à corriger, il faut être très attentif pour les éviter.

Comparer des collections différentes

Comme cela a été dit auparavant, la classe `std::string` peut être vu comme une collection de caractères (`char`). Cependant, si on essaie de comparer une variable de type `string` à un tableau de caractères, on

obtient une erreur :

main.cpp

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

int main() {
    std::string const s { "abcdef" };
    std::vector<char> const v { 'a', 'b', 'c', 'd', 'e', 'f'
};
    std::cout << std::boolalpha;
    std::cout << (s == v) << std::endl;
}
```

affiche l'erreur à la compilation suivante :

```
main.cpp:9:21: error: invalid operands to binary expression
('const std::string'
(aka 'const basic_string<char, char_traits<char>,
allocator<char> >') and
'const std::vector<char>')
    std::cout << (s == v) << std::endl;
                  ~ ^  ~
1 error generated.
```

Cette erreur signifie que le compilateur ne trouve pas d'opérateur d'égalité `==` permettant de comparer un type `std::string` avec un type `std::vector<char>`. En effet, les opérateurs de comparaison sont définis pour accepter deux arguments de même type, par exemple deux `string` ou deux `vector<char>`, mais pas deux collections de type différents, même si ces deux types correspondent tous deux à des collections de `char` (on parle parfois de typage “fort” du C++).

Dans cette situation, il est possible d'utiliser la fonction `std::equal` pour tester l'égalité de deux collections. Cet algorithme fonctionne sur des collections de types différents, à partir du moment où les éléments sont comparable par égalité. Par exemple, il sera possible de comparer des `std::string` et des `std::vector<char>` (les éléments sont dans les deux cas des `char`) ou des `std::vector<int>` et `std::vector<float>`

(il est possible de comparer un entier et un nombre réel), mais pas `std::vector<int>` et `std::vector<string>` (`int` et `string` ne sont pas comparable par égalité).

En utilisant `std::equal`, la comparaison devient :

main.cpp
```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

int main() {
    std::string const s { "abcdef" };
    std::vector<char> const v1 { 'a', 'b', 'c', 'd', 'e',
'f' };
    std::vector<char> const v2 { 'a', 'z', 'e', 'r', 't',
'y' };
    std::cout << std::boolalpha;
    std::cout << std::equal(begin(s), end(s), begin(v1), end
(v1)) << std::endl;
    std::cout << std::equal(begin(s), end(s), begin(v2), end
(v2)) << std::endl;
}
```

affiche :

```
true
false
```

Ce qui correspond bien a une comparaison des éléments un par un.

Comparer des parties de collections

Avec `std::equal`, il est possible de comparer une partie d'une collection. Pour cela, il suffit de fournir des positions différentes que le début et la fin d'une collection. Par exemple, vous pouvez incrémenter ou décrémenter les positions de début (avec `begin(s)+n`) et de fin (avec `end(s)-n`) en faisant attention de ne pas donner des valeurs en dehors

de la collection ou en utilisant des fonctions de recherche (`std::find`, que vous verrez dans un prochain chapitre).

Par exemple, pour comparer les quatre premiers éléments de deux collections, vous pouvez écrire :

main.cpp

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

int main() {
    std::string const s1 { "abcdef" };
    std::string const s2 { "abcdEF" };
    std::cout << std::boolalpha;
    std::cout << std::equal(begin(s1), end(s1), begin(s2),
end(s2)) << std::endl;
    std::cout << std::equal(begin(s1), begin(s1)+4, begin(s2),
begin(s2)+4) << std::endl;
}
```

affiche :

```
false
true
```

La première version compare la totalité des deux chaînes (`“abcdef”` et `“abcdEF”`) et retourne faux, puisque la chaîne `s2` contient des majuscules. La seconde version compare le début de la première chaîne (à partir de `begin(s1)` donc `“a”` jusqu'au quatrième caractère `begin(s1)+4` donc `“d”`) avec le début de la seconde (également `“abcd”`) et retourne vrai.

En utilisant la notation `begin(s)+n`, si vous sortez de la collection, cela produit un comportement indéterminé (*undefined behavior*). Il n'y a aucun message d'erreur vous prévenant qu'il y a un problème.
Faites bien attention de ne pas perdre de vue ce que vous comparez. Ce

n'est pas parce que vous utilisez `std::equal` (“égal” en français) que vos collection son “égales”. Seuls les éléments que vous comparez sont “egaux” (donc potentiellement des collections de tailles ou de types différents, voire des éléments de type différents). Ce n'est pas une “égalité” stricte.

Utiliser un prédicat personnalisé

Par défaut, l'algorithme `std::equal` compare chaque élément de deux collections en utilisant l’opérateur `==` de chaque élément. En pseudo-code, cela donnerait :

```
si premier élément de collection 1 est différent du premier
élément de collection 2
    alors retourner "les collections sont différentes"

si deuxième élément de collection 1 est différent du
deuxième élément de collection 2
    alors retourner "les collections sont différentes"

si troisième élément de collection 1 est différent du
troisième élément de collection 2
    alors retourner "les collections sont différentes"

...

Si tous les éléments sont identiques
    alors retourner "les collections sont identiques"
```

Lorsque les collections contiennent des éléments de types non comparables ou lorsque la comparaison d'égalité par défaut ne convient pas, il est possible de fournir un prédicat personnalisé dans `std::equal`.

Un prédicat est “quelque chose” qui prend des arguments et retourne un booléen. Vous utiliserez principalement des prédicats avec un argument (prédicat unaire) ou deux arguments (prédicat binaire) avec les algorithmes de la bibliothèque standard.

```
bool result_1 = predicat_unaire(argument_1);
```

```
bool result_2 = predicat_binaire(argument_1, argument_2);
```

Ce “quelque chose” n'est volontairement pas défini pour le moment, cela peut correspondre à différentes syntaxes que vous verrez plus tard dans ce cours (une fonction libre, une fonction lambda, un foncteur). Le plus important est donc de retenir que c'est “quelque chose” qui peut s'utiliser de la façon donnée dans le code.
Vous reconnaissez la syntaxe pour appeler une fonction. Plus généralement, il est possible d'utiliser n'importe quel objet qui peut s'utiliser comme une fonction (*callable-object*, “objet appelable”).

Ce prédicat sera utilisée à la place de l’opérateur `==` pour la comparaison des éléments un par un. En pseudo-code, cela donnerait :

```
si prédicat(premier élément de collection 1, premier élément
de collection 2)
    alors retourner "les collections sont différentes"

si prédicat(deuxième élément de collection 1, deuxième
élément de collection 2)
    alors retourner "les collections sont différentes"

si prédicat(troisième élément de collection 1, troisième
élément de collection 2)
    alors retourner "les collections sont différentes"

...

Si tous les éléments sont identiques
    alors retourner "les collections sont identiques"
```

Les objets-fonctions

La bibliothèque standard fournit quelques prédicats de base, décris dans la documentation (Function objects) et inclus dans le fichier d’en-tête

`<functional>`. Ces prédicats sont simples à comprendre, ils correspondent aux opérateurs logiques (`logical_and`, `logical_or` et `logical_not`) et de comparaison (`equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal` et `less_equal`) que vous avez déjà vu.

Attention de ne pas confondre l'algorithme `std::equal` et le prédicat `std::equal_to`.
Les autres objets-fonctions seront utiles pour les autres algorithmes de la bibliothèque standard, que vous verrez par la suite.

Les objets-fonctions peuvent être appelés comme des fonctions, mais sont avant tout des objets. Il est donc nécessaire dans un premier temps de créer l'objet avant de pouvoir l'utiliser. La création d'un objet-fonction est similaire à n'importe quelle création d'objet.

```
std::equal_to<TYPE>()
std::equal_to<>()
```

Les objets-fonctions de la bibliothèque standard sont des classes template et s'écrivent donc avec des chevrons `<>`. Il est possible de spécifier le type du prédicat entre les chevrons (ce qui produira une erreur si vous essayez d'utiliser le prédicat avec un type non compatible) ou de laisser les chevrons vides pour accepter n'importe quel type comparable.

L'objet crée peut ensuite être appelée comme une fonction.

```
const auto predicat = std::equal_to<>();
std::cout << predicat(1, 2) << std::endl;
```

Le code suivant :

main.cpp

```
#include <iostream>
#include <functional>

int main() {
    const auto predicat = std::equal_to<>();
```

```
    std::cout << std::boolalpha;
    std::cout << predicat(1, 2) << std::endl;
    std::cout << predicat(1, 1.0) << std::endl;
    std::cout << predicat('a', 'b') << std::endl;
    std::cout << predicat('a', 'a') << std::endl;
    std::cout << predicat("azerty", "abcdef") << std::endl;
    std::cout << predicat("azerty", "azerty") << std::endl;
}
```

affiche ;

```
false
true
false
true
false
true
```

Il n'est pas nécessaire de créer une variable intermédiaire pour créer un objet-fonction, mais cela permet d'avoir un code plus lisible.

Notez bien l'utilisation des parenthèses :

- la première paire pour créer l'objet-fonction ;
- la seconde pour appeler la fonction.

Si vous ne souhaitez pas créer de variable intermédiaire, il faut bien mettre les deux paires de parenthèses (cette syntaxe trouble parfois les débutants).

main.cpp

```
#include <iostream>
#include <functional>

int main() {
    std::cout << std::boolalpha;
    std::cout << std::equal_to<>()(1, 2) << std::endl;
    std::cout << std::equal_to<>()(1, 1.0) << std::endl;
    std::cout << std::equal_to<>()('a', 'b') << std::endl;
    std::cout << std::equal_to<>()('a', 'a') << std::endl;
    std::cout << std::equal_to<>()("azerty", "abcdef") <<
```

```
std::endl;
    std::cout << std::equal_to<>()("azerty", "azerty") <<
std::endl;
}
```

Les algorithmes avec prédicats personnalisés

Les prédicats peuvent être utilisés avec les algorithmes de la bibliothèque standard. Dans ce cas, les algorithmes appellent directement le prédicat sur les éléments d'une collection. Cela implique qu'il faut donner un objet directement appelable, donc qu'il faut instancier l'objet-fonction avant de la passer en argument d'un algorithme.

Pour tous les algorithmes de la bibliothèque standard qui acceptent un prédicat, celui-ci est donné en dernier argument. Par exemple, avec `std::equal`, la syntaxe devient :

```
std::equal(itérateur, itérateur, itérateur, itérateur)
std::equal(itérateur, itérateur, itérateur, itérateur,
PRÉDICAT)
```

Il est possible de créer une variable intermédiaire pour le prédicat ou de l'instancier directement dans l'appel de l'algorithme.

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

int main() {
    const std::vector<int> v { 1, 2, 3, 4, 5 };
    const std::vector<int> w { 2, 4, 3, 1, 5 };

    std::cout << std::boolalpha;
    std::cout << std::equal(begin(v), end(v), begin(v), end(
v), std::equal_to<>()) << std::endl;
    std::cout << std::equal(begin(v), end(v), begin(w), end(
w), std::equal_to<>()) << std::endl;
```

```
    std::cout << std::equal(begin(v), end(v), begin(v), end(
v), std::not_equal_to<>()) << std::endl;
    std::cout << std::equal(begin(v), end(v), begin(w), end(
w), std::not_equal_to<>()) << std::endl;
}
```

affiche :

```
true
false
false
false
```

Exercice

Est-ce que les deux syntaxes suivantes donnent le même résultat ? C'est-à-dire de comparer si deux collections sont différentes. (Notez bien l'opérateur de négation logique `!` dans la seconde ligne).

```
bool result_1 =   std::equal(begin(v), end(v), begin(w), end
(w), std::not_equal_to<>());
bool result_2 = ! std::equal(begin(v), end(v), begin(w), end
(w));
```

L'ordre lexicographique

L'égalité et l'inégalité de deux collections est simple a définir : deux collections sont égales si elles contiennent les mêmes éléments (même nombre d'éléments, dans le même ordre). Elles sont différentes dans le cas contraire.

Mais quel sens donner à la phrase "une collection est inférieure à une autre collection" ?

L'ordre lexicographique est une méthode qui permet de comparer deux collections. Même si vous ne connaissez pas le terme, vous connaissez obligatoirement cette méthode : c'est celle qui est utilisée pour trier des mots, en particulier dans un dictionnaire. (N'oubliez pas qu'une chaîne de caractères peut être vue comme une collection de caractères).

Pour rappel, voici comment appliquer cette méthode :

- Commencez par prendre le premier élément de chaque collection et comparez les.
- Si l'un des éléments d'une des collections est inférieur à l'élément de l'autre collection, la collection correspondante est inférieure à l'autre.
- Si les deux éléments sont égaux, comparez les éléments suivants de chaque collection.
- Si une collection se termine avant l'autre, elle est inférieure àl'autre.
- Si tous les éléments sont identiques, les collections sont égales.

Voici un exemple pour bien comprendre, avec les chaînes “abc” et “acd”. Les premiers caractères sont “a” et “a”. Ils sont identiques, passez aux deuxièmes caractères. Ceux-ci sont “b” et “c”. Comme “b” est plus petit que “c”, la chaîne “abc” est inférieure à la chaîne “acd”.

Un autre exemple, comparez “abc” et ab”. Les premiers et deuxièmes caractères sont identiques, il faudrait donc comparer les troisièmes caractères. Cependant, la seconde chaîne ne possède pas de troisième caractère, elle est donc inférieure à la première.

De même avec une collection d'entiers. Par exemple, la collection `{ 1, 2, 3 }` sera inférieure à la collection `{ 1, 2, 4 }` et supérieure à la collection `{ 1, 2 }`.

main.cpp

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

int main() {
    std::cout << std::boolalpha;
    std::cout << (std::string { "abc" } < std::string { "acd"
}) << std::endl;
    std::cout << (std::string { "abc" } < std::string { "ab"
}) << std::endl;
    std::cout << (std::vector<int> { 1, 2, 3 } < std::vector
```

```
<int> { 1, 2, 4 }) << std::endl;
    std::cout << (std::vector<int> { 1, 2, 3 } < std::vector
<int> { 1, 2 }) << std::endl;
}
```

affiche :

```
true
false
true
false
```

La comparaison “plus petit que” est également un concept (LessThanComparable), ce qui implique que différentes propriétés doivent être respectées :

- **identité** : une collection n'est pas inférieure à elle-même (elle est égale à elle-même) ;
- **inverse** : si une collection est inférieur à une seconde collection, la seconde collection n'est pas inférieure à la première (elle est supérieure ou égale) ;
- **transitivité** : si une collection a est inférieur à une collection b et que cette collection b est inférieure à une collection c, alors la collection a est inférieure à la collection c.

Ce concept “LessThanComparable” est valable pour d'autres types du C++ (`int`, `float`, etc.) et de la bibliothèque standard (`std::string`, `std::complex`, `std::vector`, etc.).

Lorsqu'une classe définie un opérateur de comparaison, il est logique que les autres opérateurs soient aussi définis (il faudrait que cela ait un sens de ne pas les définir). Vous apprendrez dans la partie sur la programmation orientée objet (en particulier dans la partie sur la sémantique de valeur) comment définir ces opérateurs dans une classe que vous créez.

L'algorithme std::lexicographical_compare

L'algorithme `std::equal` permet de comparer si deux collections sont égales ou différentes. L'équivalent pour la comparaison est l'algorithme `std::lexicographical_compare`. Tout comme `std::equal` est une version plus générique de l'opérateur `==` (et indirectement de `!=`), l'algorithme `std::lexicographical_compare` est une version plus générique de l'opérateur `<` (et donc indirectement des opérateurs `>`, `<=` et `>=`). En particulier, `std::lexicographical_compare` pourra être utilisé sur des sous-collections ou des collections de types différents.

La fonction `std::lexicographical_compare` (documentation) prend comme arguments le début et la fin des deux collections à comparer. Elle existe en deux versions, avec ou sans prédicat personnalisé, et retourne vrai si la première collection est inférieure à la seconde.

main.cpp

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

int main() {
    std::string const s1 { "azerty" };
    std::string const s2 { "abcdef" };
    std::string const s3 { "qwerty" };
    std::cout << std::boolalpha;
    std::cout << std::lexicographical_compare(begin(s1), end
(s1), begin(s2), end(s2)) << std::endl;
    std::cout << std::lexicographical_compare(begin(s1), end
(s1), begin(s3), end(s3)) << std::endl;
}
```

affiche :

```
false
true
```

Exercices

- comparer vector<int> et vector<float>
- comparer vector<int> et vector<string> (Aide : utilisez `std::stoi` dans une fonction lambda)
- tester si une chaîne est un palindrome (un palindrome est un mot qui peut être lu de droite à gauche ou de gauche à droite, comme par exemple “kayak” ou “radar”).

main.cpp

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

int main() {
    std::string const s1 { "azerty" }; // non palindromique
    std::string const s2 { "abccba" }; // palindromique
    std::cout << std::boolalpha;
    std::cout << std::equal(begin(s1), end(s1), s1.rbegin())
<< std::endl;
    std::cout << std::equal(begin(s2), end(s2), s2.rbegin())
<< std::endl;
}
```

affiche :

```
false
true
```

- Tester des collections différentes

main.cpp

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

int main() {
    std::string const s { "abcdef" };
```

```
    std::vector<char> const a { 'a', 'z', 'e', 'r', 't', 'y'
};
    std::cout << std::boolalpha;
    std::cout << std::lexicographical_compare(begin(s), end(
s), begin(a), end(a)) << std::endl;
}
```

affiche :

```
true
```

- Comparer des parties de collections

main.cpp

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

int main() {
    std::string const s1 { "abcdef" };
    std::string const s2 { "abcdfg" };
    std::cout << std::boolalpha;
    std::cout << std::lexicographical_compare(begin(s1), end
(s1), begin(s2), end(s2)) << std::endl;
    std::cout << std::lexicographical_compare(begin(s1),
begin(s1)+4, begin(s2), begin(s2)+4) << std::endl;
}
```

affiche :

```
true
false
```

- Utiliser un prédicat personnalisé

main.cpp

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

```
int main() {
    std::string const s1 { "ABCDEF" };
    std::string const s2 { "azerty" };
    std::cout << std::boolalpha;
    std::cout << std::lexicographical_compare(begin(s1), end
(s1), begin(s2), end(s2)) << std::endl;
    std::cout << std::equal(begin(s1), end(s1), begin(s2),
end(s2),
        [](auto lhs, auto rhs){ return std::toupper(lhs) <
std::toupper(rhs); })
        << std::endl;
}
```

affiche :

```
true
false
```

Chapitre précédent	Sommaire principal	Chapitre suivant