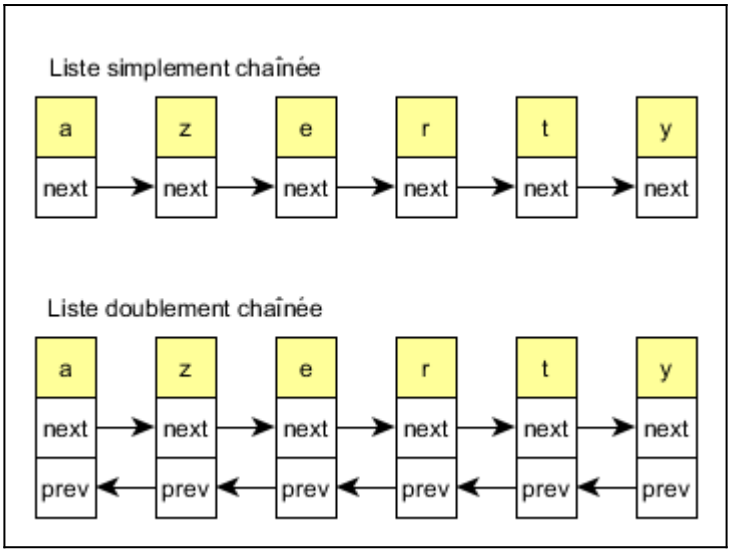
Les catégories d'itérateurs

Chaque type de collection ne propose pas les mêmes fonctionnalités en termes de parcours des éléments. Par exemple, avec une liste doublement chaînée, il sera possible d'obtenir l'élément précédent un autre élément, alors qu'avec une liste simplement chaînée, cela ne sera pas possible (il faudra parcourir la collection depuis le début pour trouver cet élément précédent).



Les fonctionnalités disponibles ont forcément un impact sur ce que vous pouvez faire avec un itérateur. Pour gérer cela, les itérateurs sont organisés par catégories, chaque type d'itérateur reprenant les fonctionnalités de la catégorie précédente en ajoutant de nouvelles fonctionnalités. Par exemple, la catégorie `InputIterator` propose la fonctionnalité “élément suivant”, la catégorie `BidirectionalIterator` propose les fonctionnalités de `InputIterator` et la fonctionnalité

“élément précédent”.

Chaque collection propose donc un type d'itérateur. En connaissant ce type, vous pouvez connaître les fonctionnalités qui seront utilisables ou non.

La liste des catégories est donnée dans la page de documentation : Iterator library.

Il existera une nouvelle catégorie dans le C++17 : `ContiguousIterator`. Elle correspond à des collections dont les éléments sont contigus en mémoire. Pour les versions antérieures du C++, l'équivalent est la catégorie `RandomAccessIterator` avec quelques classes, comme par exemple `std::vector`, `std::array` ou `std::string`.

InputIterator et OutputIterator

Ces deux catégories proposent les fonctionnalités de base, que vous avez déjà vues :

- le concept “élément suivant”, qui permet d'obtenir l'élément suivant d'un élément donné ;
- le concept “déréférencement”, qui permet d'accéder à l'élément correspondant à l'itérateur.

Ces deux concepts, associés aux concepts “début” (`begin`) et “fin” (`end`) des collections sont les concepts de base permettant d'écrire la majorité des algorithmes généralistes. Lorsque vous créez un nouveau type de collection ou d'itérateur, il faudra fournir ces services de base pour avoir la meilleure réutilisabilité possible.

Le concept "comparable par égalité"

Vous connaissez déjà ce concept, cela signifie qu'il est possible de comparer par égalité (et donc par inégalité aussi) deux itérateurs, en utilisant l'opérateur `==` (respectivement l'opérateur `!=`). Vous avez en particulier vu cela pour vérifier si un itérateur est valide, en le comparant avec l'itérateur retourné par `end`.

```
vector<int> v { ... };
const auto v_is_empty = (begin(v) == end(v)); // type bool
```

Cette opération est la seule que vous pouvez faire avec un itérateur invalide (par exemple celui retourné par `end` ou après qu'une collection soit modifiée). Toutes les autres opérations sur un itérateur invalide provoquent un comportement indéterminé et ne doivent être jamais faites.

Le concept "élément suivant"

Ce concept correspond à plusieurs syntaxes : avec l'opérateur unaire `++` (en pré ou post-fixé, c'est-à-dire placé avant ou après la variable) ou la fonction `std::next`.

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = cbegin(v);
    std::cout << (*it) << std::endl;
    ++it;
    std::cout << (*it) << std::endl;
    it++;
    std::cout << (*it) << std::endl;
    it = std::next(it);
    std::cout << (*it) << std::endl;
}
```

affiche :

```
1
2
3
4
```

Le plus courant est d'utiliser l'opérateur `++` préfixé (`++it`), utilisez cette syntaxe par défaut.

Et n'oubliez pas que vous ne pouvez faire cela que si l'itérateur est différent de `end`. En toute rigueur, il faudrait donc ajouter des assertions au code précédent pour vérifier la validité de l'itérateur après chaque modification et avant chaque utilisation :

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = cbegin(v);
    assert(it != cend(v));
    std::cout << (*it) << std::endl;
    ++it;
    assert(it != cend(v));
    std::cout << (*it) << std::endl;
    it++;
    assert(it != cend(v));
    std::cout << (*it) << std::endl;
    it = std::next(it);
    assert(it != cend(v));
    std::cout << (*it) << std::endl;
}
```

Dans un code aussi simple que le précédent, ajouter ces assertions n'est pas nécessaire, mais dans un code réel, cette vérification est importante.

S'il est possible de parcourir une collection à partir d'un élément jusqu'à un autre élément, il est donc possible de compter le nombre d'éléments entre deux éléments. La fonction `std::distance` prend en paramètre

deux itérateurs dans une collection et retourne le nombre de fois qu'il faut appeler `std::next` pour passer du premier au second.

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = cbegin(v);             // InputIterator
    ++it;
    ++it;
    ++it;
    std::cout << std::distance(cbegin(v), it) << std::endl;
    std::cout << std::distance(it, cend(v)) << std::endl;
}
```

affiche :

```
3
1
```

Le comportement de `std::distance` est indéterminé si le second itérateur n'est pas accessible à partir du premier itérateur en appelant `std::next`.

Le concept "déréférencer un itérateur"

Un itérateur étant une indirection, il est nécessaire de pouvoir accéder à l'élément correspond à un itérateur. Vous avez vu dans le chapitre précédent qu'il faut utiliser l'opérateur unaire préfixé `*` pour cela (à ne surtout pas confondre avec l'opérateur binaire de multiplication `*`).

main.cpp

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

int main() {
```

```
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = cbegin(v);
    std::cout << (*it) << std::endl;
}
```

affiche :

```
1
```

Les catégories `InputIterator` et `OutputIterator` se distinguent par le fait de pouvoir modifier la valeur (itérateur non constant) ou non (itérateur constant). Avec `InputIterator` (obtenu par exemple avec `cbegin` ou `cend`), vous ne pouvez pas modifier l'élément correspondant à un itérateur (itérateur constant). Avec `OutputIterator`, vous pouvez modifier l'élément.

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto cit = cbegin(v);             // InputIterator
    (*cit) = 0;                       // erreur
    std::cout << (*cit) << std::endl;
    auto it = begin(v);               // OutputIterator
    (*it) = 0;                         // ok
}
```

produit l'erreur :

```
main.cpp:7:12: error: cannot assign to return value because
function 'operator*' returns a const value
    (*cit) = 0;                       // erreur
    ~~~~~~ ^
```

ForwardIterator

La catégorie `ForwardIterator` ajoute la fonctionnalité “avancer de plusieurs éléments” aux catégories de base `InputIterator` et

`OutputIterator`. Il est bien sûr possible d'avancer de plusieurs éléments dans ces deux dernières catégories, mais il faut pour cela appliquer plusieurs fois une même opération “élément suivant”. Avec `ForwardIterator`, cela est réalisé en une seule opération.

Un exemples de collection proposant des itérateurs de type `ForwardIterator` est `std::forward_list`.

Encore une fois, il existe plusieurs syntaxes pour réaliser cette opération : avec les opérateurs `+` et `+=` et avec la fonction `std::advance`. Dans tous les cas, ces opérations prennent en paramètre un itérateur et une valeur entière correspondant au nombre d'éléments à avancer.

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = cbegin(v);
    it += 2;
    std::cout << (*it) << std::endl;
    it = cbegin(v);
    it = it + 1;
    std::cout << (*it) << std::endl;
    it = cbegin(v);
    advance(it, 3);
    std::cout << (*it) << std::endl;
}
```

affiche :

```
3
2
4
```

Ces différents opérations ne vérifient pas les accès en dehors de la collection, ce qui peut provoquer un comportement indéterminé. C'est (encore une fois) au développeur de vérifier que l'appel à ces opérations ne sont pas invalides.

BidirectionalIterator

La catégorie ajoute à la catégorie `ForwardIterator` le concept d'élément précédent. Sans surprise, il existe aussi plusieurs syntaxes, qui sont l'équivalent symétrique des syntaxes pour “élément suivant”, avec les opérateurs `--` (pré et post-fixé) et la fonction `std::prev` (pour *previous*).

Un exemples de collection proposant des itérateurs de type `BidirectionalIterator` est `std::list`.

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = cend(v);
    --it;
    std::cout << (*it) << std::endl;
    it--;
    std::cout << (*it) << std::endl;
    it = prev(it);
    std::cout << (*it) << std::endl;
}
```

affiche :

```
4
3
2
```

Il faut également vérifier que ces opérations sont valides, en essayant d'accéder avant le premier élément.

RandomAccessIterator

La dernière catégorie d'itérateur permet un accès “aléatoire” aux

éléments d'une collection. Par “aléatoire”, il ne fait pas comprendre “au hasard”, mais “n'importe où”. Par exemple, si une collection contient cinq éléments, il est possible d'accéder directement au quatrième élément, puis au deuxième, puis au cinquième, etc.

Des exemples de collection proposant des itérateurs de type `RandomAccessIterator` sont `std::vector` ou `std::array`.

L'accès à un élément quelconque est réaliser en utilisant l'opérateur `[]` post-fixé. Cet opérateur prend en argument l'indice de l'élément dans la collection. En C++, les indices commencent à zéro et finissent à `size() - 1`.

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    const auto it = cbegin(v);
    std::cout << it[3] << std::endl;
    std::cout << it[0] << std::endl;
    std::cout << it[2] << std::endl;
}
```

affiche :

```
4
1
3
```

Il est plus courant d'utiliser l'opérateur `[]` directement sur une collection pour obtenir une valeur, plutôt que d'utiliser cet opérateur sur un itérateur.

Dans tous les cas, il est important de vérifier que les accès sont valide (c'est-à-dire que l'indice est compris entre zéro et `size() - 1`).

```
// accès via un itérateur
assert(it[n] < v.size());
```

```
auto value = *(it[n]);

// accès via la collection
assert(n < v.size());
auto value = v[n];
```

Pour terminer, les itérateurs de type `RandomAccessIterator` sont “comparable par plus petit que” (`LessThanComparable`). Cela signifie que vous pouvez utiliser les opérateurs de comparaison `<`, `>`, `<=` et `>=`.

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    const auto it = cbegin(v) + 3;
    std::cout << std::boolalpha << (it < end(v)) << std::
endl;
}
```

affiche :

```
true
```

Adaptateurs d'itérateurs

Itérator reverse

En complément des catégories d'itérateurs disponibles pour chaque collection, la bibliothèque standard fournit également des adaptateurs qui permettent de modifier le comportement des itérateurs précédents. Par exemple, `reverse_iterator` permet de transformer un itérateur en itérateur “reverse” (qui permet de parcourir une collection dans le sens inverse).

Chaque adaptateur se compose d'un modificateur de type et d'une

fonction de conversion. Le modificateur permet de créer une nouveau type d'itérateur à partir d'un type d'itérateur. Par exemple, pour `std::vector` avec `std::reverse_iterator` :

```
using forward_it = std::vector<int>::iterator;
using reverse_it = std::reverse_iterator<forward_it>;
```

En pratique, il est plus simple d'utiliser l'inférence de type et laisser le compilateur déterminer automatiquement le type exact d'itérateur.

La fonction de conversion prend en argument un itérateur et retourne un itérateur de type “reverse” correspondant au même élément. N'oubliez pas que le sens de parcours est inversé, ce qui implique qu'il faut également inverser l'ordre des arguments dans les fonctions (`begin` devient `rend` et `end` devient `rbegin`). Par exemple, pour créer un itérateur “reverse” avec la fonction `std::make_reverse_iterator` :

main.cpp

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

int main() {
    const std::string s { "abcdefgh" };
    const auto first = begin(s) + 1;
    const auto last = end(s) - 1;
    std::cout << std::string(first, last) << std::endl;
    const auto rfirst = std::make_reverse_iterator(last);
    const auto rlast = std::make_reverse_iterator(first);
    std::cout << std::string(rfirst, rlast) << std::endl;
}
```

affiche :

```
bcdefg
gfedcb
```

Bien sûr, si vous récupérez un itérateur en utilisant `begin` ou `end`, il est plus simple d'utiliser directement `rbegin` et `rend`. Mais un itérateur peut également être obtenu en utilisant un algorithme, comme par exemple `std::find`. Vous pouvez souhaitez réaliser cette recherche dans le sens

direct (donc à partir des itérateurs retournés par `begin` et `end`). Puis appliquer un second algorithme dans le sens inverse en utilisant `std::make_reverse_iterator`.

Insertion et flux

Jusqu'à maintenant, les itérateurs que vous avez vu permettent de lire ou modifier des éléments existants dans des collections. Cela implique qu'il est nécessaire d'initialiser les collections pour qu'elles contiennent suffisamment d'éléments pour appliquer les algorithmes. Par exemple, dans le code suivant, il faut créer des éléments dans la collection de destination avant de faire la copie :

```
const std::vector<int> in { 1, 2, 3, 4, 5 };
std::vector<int> out(in.size());
std::copy(begin(in), end(in), begin(out));
```

Les adaptateurs d'itérateurs de type “inserter” permettent d’appeler les fonctions d'ajout d'éléments dans une collection `push_back`, `push_front` et `insert`. De cette manière, il est possible d'augmenter le nombre d'éléments dans un collection.

Lorsque vous insérer des éléments dans une collection, cela peut produire des ré-allocations de la mémoire utilisée par ces collections. Cela peut potentiellement avoir un coût important en termes de performances et invalider les itérateurs existants. Il est possible d'utiliser la fonction `reserve` pour pré-allouer la mémoire et limiter ce problème.
Les fonctions permettant d'insérer dans une collection sont les suivantes :

- `front_inserter` pour insérer au début d'une collection (le type correspondant est `front_insert_iterator`) ;
- `inserter` pour insérer n'importe où dans une collection (le type

correspondant est `insert_iterator`) ;
- `back_inserter` pour insérer à la fin d'une collection (le type correspondant est `back_insert_iterator`).

Le code suivant montre l'utilisation de ces fonctions (avec une collection de type `std::list`, qui permet d'utiliser les trois types d'insertion, alors que `std::vector` ne permet pas d'insérer au début de la collection) :

main.cpp

```
#include <iostream>
#include <list>
#include <algorithm>

int main() {
    const std::list<char> in { 'a', 'b', 'c', 'd' };
    std::list<char> out { '1', '2', '3', '4' };
    std::copy(begin(in), end(in), std::front_inserter(out));
    for (auto c: out) std::cout << c;
    std::cout << std::endl;

    out = { '1', '2', '3', '4' };
    const auto it = std::find(begin(out), end(out), '3');
    std::copy(begin(in), end(in), std::inserter(out, it));
    for (auto c: out) std::cout << c;
    std::cout << std::endl;

    out = { '1', '2', '3', '4' };
    std::copy(begin(in), end(in), std::back_inserter(out));
    for (auto c: out) std::cout << c;
    std::cout << std::endl;
}
```

affiche :

```
dcba1234
12abcd34
1234abcd
```

La collection de sortie contient au début “1234”. Dans la première ligne du résultat, l'algorithme `std::copy` commence par ajouter “a” (le

résultat est donc “a1234”), puis “b” (le résultat est ensuite “ba1234”) et ainsi de suite, jusqu'à obtenir “dcba1234”.

Exercice : comment modifier la ligne de code avec le premier `std::copy` pour obtenir comme résultat “abcd1234” ?

Pour la deuxième ligne du résultat, `std::inserter` prend en argument un itérateur sur la position à utiliser pour l'insertion. Dans cet exemple, l'itérateur correspond à l'élément “3”, l'insertion se fera à cette position. L'algorithme commence par copier “a” dans “1234” (le résultat est “12a34”), puis “b” (le résultat est “12ab34”) et ainsi de suite.

Pour la dernière ligne du résultat, les caractères “a”, “b”, “c” et “d” sont insérés à la fin de la liste “1234”, pour obtenir “1234abcd”.

De la même manière, il existe des adaptateurs permettant de travailler sur les flux et des tampons de mémoire (*buffer*), en utilisant les opérateurs `«` ou `»`. Cela permet d'écrire ou lire directement dans un flux, comme par exemple un fichier ou `std::cout`. (Un tampon mémoire est simplement un flux en mémoire. Généralement, les flux standards comme `std::cout` et les fichiers utilisent en interne des tampons mémoires, pour des raisons de performance. Cela sera détaillé dans les chapitres sur les entrées-sorties).

Pour les flux et les tampons de mémoire (*buffer*), les fonctions sont les suivantes :

- `istream_iterator` pour utiliser un flux d'entrée `istream` (avec l'opérateur `»`) ;
- `ostream_iterator` pour utiliser un flux de sortie `ostream` (avec l'opérateur `«`) ;
- `istreambuf_iterator` pour utiliser un tampon mémoire d'entrée (avec l'opérateur `»`) ;
- `ostreambuf_iterator` pour utiliser un tampon mémoire de sortie (avec l'opérateur `«`).

Par exemple, pour créer un itérateur `std::ostream_iterator` sur `std::cout` :

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    std::copy(begin(v), end(v), std::ostream_iterator<int>(
std::cout, " - "));
}
```

affiche :

```
1 - 2 - 3 - 4 -
```

L'utilisation des autres adaptateurs d'itérateurs de flux est similaire.

Chapitre précédent	Sommaire principal	Chapitre suivant