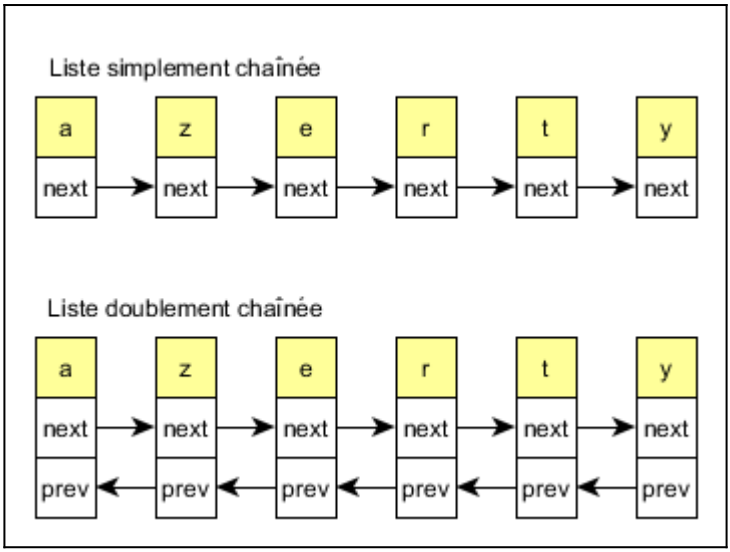
Les catégories d'itérateurs

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 forcement 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.

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*).

<code cpp main.cpp> #include <iostream> #include <vector>

int main() {

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