

Les fonctionnalités de base des collections

Le chapitre précédent présentait chaque collection et ses spécificités, mais sans entrer dans les détails. Certaines fonctionnalités étant communes à plusieurs collections, ces fonctionnalités ne vont pas être décrites par collection, mais par catégories de fonctionnalités. N'hésitez pas à consulter la documentation pour savoir si une collection en particulier permet d'utiliser une fonctionnalité ou non.

Certaines fonctionnalités ont déjà été décrites dans les chapitres précédents, ce chapitre ne sera qu'un rappel dans ce cas.

Sémantique de collection et des éléments

Vous avez déjà utilisé à plusieurs reprises des collections dans les chapitres précédents. Vous avez par exemple vu qu'une collection pouvait être comparée par égalité (`std::equal`), être triée (en utilisant la comparaison "plus petit que" `std::less`) ou être copiée (`std::copy`).

Les fonctionnalités proposées par les collections dépendent des fonctionnalités possibles des éléments qu'elle contient. Par exemple, copier une collection signifie en fait copier les éléments un par un. Pour copier une collection, il faut donc que les éléments soient copiables.

main.cpp

```
#include <vector>
#include <memory>

int main() {
    using copiable_collection = std::vector<std::shared_ptr<
int>>;
    copiable_collection cc1 {};
    copiable_collection cc2 = cc1;
```

```
using noncopiable_collection = std::vector<std::  
unique_ptr<int>>;  
noncopiable_collection uc1 {};  
noncopiable_collection uc2 = uc1;  
}
```

(Ne vous inquiétez pas pour `std::shared_ptr` et `std::unique_ptr`, vous verrez ces classes plus tard. Elles sont utilisées dans cet exemple uniquement parce que la première est copiable et pas la seconde.)

Dans ce code, avec une classe copiable (`std::shared_ptr`), la copie sera autorisée (`cc2 = cc1`). Au contraire, avec une classe non copiable (`std::unique_ptr`), essayer de copier de collection (`uc2 = uc1`) produira une erreur de compilation.

```
/usr/local/include/c++/5.3.0/bits/unique_ptr.h:356:7: note:  
declared here  
    unique_ptr(const unique_ptr&) = delete;  
    ^
```

(Le message d'erreur ne dit pas explicitement que `std::unique_ptr` n'est pas copiable, il indique qu'une fonction particulière, le constructeur par copie, est supprimée. Les messages d'erreur en C++ sont parfois difficiles à comprendre, cela fait partie de l'apprentissage du C++ d'apprendre à les comprendre.)

N'oubliez pas que les fonctionnalités décrites dans ce chapitre dépendent donc du type d'élément que vous utiliserez dans une collection.

Créer une collection

Construction par défaut

Toutes les collections de la bibliothèque standard sont constructibles par défaut ([DefaultConstructible](#)), c'est-à-dire sans aucun paramètre ou de liste de valeurs.

```
std::vector<int> v {};  
std::list<double> l {};  
std::map<char, std::string> m {};
```

Syntaxe alternative (rappel)

Pour rappel, il existe d'autres syntaxes possibles pour créer une variable par défaut, qui ne sont pas recommandées, mais que vous pouvez rencontrer dans un ancien code C++.

```
std::vector<int> v; // sans accolades
```

Autre rappel, il est classique de faire l'erreur d'utiliser des parenthèses. Cependant, cela ne permet pas de créer une collection par défaut. (Cela permet en fait de déclarer une fonction.)

```
std::vector<int> v(); // erreur, ce code déclare une  
fonction
```

Liste de valeurs

La méthode la plus simple pour initialiser une collection avec des valeurs est de fournir celles-ci directement lors de l'initialisation.

```
const std::vector<int> v { 1, 2, 3, 4 };
```

Les valeurs doivent être de même type que la collection ou être implicitement convertibles.

```
// conversion implicite de int en double  
const std::vector<double> v { 1, 2, 3, 4 };
```

Pensez à utiliser `const` si la collection n'est pas modifiée par la suite.

Il est également possible d'utiliser une liste de valeurs sur une variable existante, par affectation.

```
v = { 1, 2, 3, 4, 5 };
```

Lorsque toutes les valeurs sont identiques (ou lorsque vous souhaitez initialiser avec de nombreuses valeurs), vous pouvez indiquer la taille de la collection à la création, ainsi qu'une valeur par défaut optionnelle.

```
std::vector<int> u (5); // u = { 0, 0, 0, 0, 0 }  
std::vector<int> v (5, 12); // v = { 12, 12, 12, 12, 12 }
```

Notez bien l'utilisation des parenthèses dans ce code. Cette syntaxe peut être ambiguë, faites bien attention de ne pas les confondre.

```
v(5) // initialisation avec 5 éléments valant 0  
v{5} // initialisation avec 1 élément valant 5
```

Copie et déplacement

Lors de la déclaration

Les collections sont également copiables (*Copy*) et déplaçables (*Move*), par construction (*Constructible*) et affectation (*Assignable*). Ce qui fait que les collections peuvent être [CopyConstructible](#), [MoveConstructible](#), [CopyAssignable](#) ou [MoveAssignable](#).

Lorsqu'une collection est copiée dans une autre collection, cela implique que chaque élément de la première collection va être copié dans la seconde collection. Finalement, les deux collections contiendront la même liste d'éléments.

Lorsqu'une collection est déplacée dans une autre collection (par exemple en utilisant la fonction `std::move`, qui se traduit par "déplacer"), cela signifie que les éléments sont retirés de la première collection pour être déplacés dans la seconde. Finalement, la première collection sera vide et la seconde contiendra les éléments qui se trouvaient précédemment dans la première collection.

main.cpp

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

int main() {
    std::vector<int> v1 { 1, 2, 3 };

    const std::vector<int> v2 = v1;           // copie
    std::cout << v1.size() << ' ' << v2.size() << std::endl;

    const std::vector<int> v3 = std::move(v1); //
    déplacement
    std::cout << v1.size() << ' ' << v3.size() << std::endl;
}
```

affiche :

```
3 3
0 3
```

Après la copie, les collections `v1` et `v2` contiennent tous les deux trois éléments. Après le déplacement, la collection `v1` ne contient plus d'élément, alors que `v3` contient trois éléments.

Les notions de copie et déplacement d'objets sont très importantes en C++, en particulier pour la gestion de la durée de vie des objets. Cela sera approfondi dans un chapitre dans la suite de ce cours.

État d'un objet après déplacement

Dans l'exemple précédent, le tableau `v1` ne contient plus d'élément après le déplacement. En fait, en toute rigueur, ce n'est pas forcément vrai. Après un déplacement, la norme C++ requiert qu'un objet soit valide, mais il peut être dans n'importe quel état. Il peut donc être vide ou ne pas être modifié.

En pratique, les compilateurs majeurs vident la collection dans ce cas. Mais ce n'est pas une garantie.

Affectation

En plus de la copie et déplacement par construction, les collections peuvent être copiées et déplacées par affectation.

main.cpp

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

int main() {
    std::vector<int> v1 { 1, 2, 3 };

    std::vector<int> v2; // déclaration
    v2 = v1;           // affectation par copie
    std::cout << v1.size() << ' ' << v2.size() << std::endl;

    std::vector<int> v3; // déclaration
    v3 = std::move(v1); // affectation par déplacement
    std::cout << v1.size() << ' ' << v3.size() << std::endl;
}
```

affiche :

```
3 3
0 3
```

Faites bien attention à distinguer la déclaration et l'affectation. La première permet de créer une nouvelle variable, la seconde permet de modifier une variable existante.

Pour les reconnaître, pensez à regarder si un type est défini ou non devant le nom de la variable. Avec un type, c'est une déclaration, sans type, c'est une affectation.

```
TYPE variable = valeur; // déclaration
variable = valeur; // affectation
```

C'est une des raisons qui justifie de préférer la déclaration avec des accolades plutôt qu'une signe égal.

```
TYPE variable {}; // ok
variable {}; // invalide
```

Notez aussi qu'utiliser l'affectation implique que les variables ne peuvent pas être déclarées comme constantes (ce qui peut avoir un impact sur le travail du compilateur). Il est donc recommandé de déclarer aussi souvent que possible ses variables avec `const`, ce qui implique de les déclarer le plus tard possible, c'est-à-dire juste avant de les utiliser, lorsque vous avez toutes les informations pour les initialiser lors de la déclaration.

Avec des itérateurs

Une syntaxe alternative que vous avez déjà rencontré est l'utilisation d'une paire d'itérateurs pour copier (par défaut) ou déplacer (en utilisant `std::make_move_iterator`, qui sera détaillé dans le chapitre sur les catégories d'itérateurs) les éléments d'une collection dans une autre collection.

Les éléments peuvent être ajoutés lors de la création en les fournissant lors de la construction ou sur une variable existante en utilisant la fonction membre `assign`.

```
const std::vector<int> u { 1, 2, 3, 4 };

// lors de la creation
const std::vector<int> v (begin(u), end(u));

// fonction assign
std::vector<int> w;
w.assign(begin(u), end(u));
```

Notez bien encore une fois l'utilisation des parenthèses au lieu des accolades.

Les éléments existants au préalable dans la collection sont supprimés.

La fonction membre `assign` est équivalente à une affectation d'une

nouvelle collection, mais sans devoir créer une collection intermédiaire.

```
w.assign(begin(u), end(u));  
  
// est équivalent à  
  
w = std::vector<int>(begin(u), end(u));
```

Les avantages de l'utilisation de paire d'itérateurs sur la copie directe est que cela permet d'initialiser une collection en n'utilisant qu'une partie des éléments d'une autre collection. De plus, cela permet d'utiliser des éléments provenant de collections de types différents (du moment que les éléments sont convertibles).

main.cpp

```
#include <vector>  
#include <iostream>  
  
int main() {  
    const std::vector<int> u { 1, 2, 3, 4 };  
    const std::vector<double> v { u };           //  
    erreur  
    const std::vector<double> w (begin(u), end(u)); // ok  
}
```

Dans le premier cas, le compilateur essaie de trouver une conversion possible entre `std::vector<int>` et `std::vector<double>`, c'est-à-dire qu'il essaie de trouver un type quelconque `T` qui puisse servir d'intermédiaire :

```
const T u_temp { u };  
std::vector<double> v { u_temp };
```

Il n'existe pas un tel type et la compilation échoue.

Dans le second cas, le compilateur essaie de convertir les éléments un par un. Comme `int` est implicitement convertible en `double`, cette syntaxe compile.

Plus généralement, lorsque vous avez un type template (en particulier

une collection de la bibliothèque standard), une instance particulière, par exemple `T<A>`, ne sera généralement pas convertible en une autre instance `T`.

Taille et capacité

La fonction membre `size` permet de connaître le nombre d'éléments contenus dans une collection.

```
const std::vector<int> v { 1, 2, 3, 4 };  
std::cout << v.size() << std::endl; // affiche 4
```

Pour tester si une collection est vide, il est possible de vérifier que le nombre d'éléments vaut zéro, mais il est plus simple d'utiliser la fonction `empty`.

```
v.empty(); // équivalent à (v.size() == 0)
```

Il est possible de modifier directement la taille d'une collection en utilisant la fonction membre `resize` (redimensionner), avec ou sans valeur par défaut.

```
v.resize(10); // v contient 10 éléments valant 0  
v.resize(15, 12); // v contient 15 fois la valeur 12
```

Si une collection est redimensionnée avec une taille plus petite, les éléments en fin de collection sont simplement supprimés. Pour supprimer tous les éléments, vous pouvez utiliser la fonction `clear` (nettoyer).

Ces quatre fonctions sont disponibles pour toutes les collections. En revanche, les fonctions suivantes sont uniquement disponibles pour `std::vector` et les chaînes de caractères (`std::string` et équivalents).

Réserve

Comme vous l'avez vu dans le chapitre précédent, pour des raisons de performances, ces collections peuvent réserver plus de mémoire qu'elles en ont réellement besoin. Par exemple, un `std::vector` peut contenir cinq éléments, mais réserver la mémoire pour dix éléments. Ainsi, lorsque vous ajoutez un sixième élément, le `std::vector` n'a pas besoin d'ajouter cet élément en mémoire. Il ne fait qu'utiliser l'un des éléments de la réserve.

La fonction membre `capacity` (capacité) permet de connaître la taille totale réservée par une collection.

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    std::cout << "size: " << v.size() << std::endl;
    std::cout << "capacity: " << v.capacity() << std::endl;
    v.push_back(5);
    std::cout << "size: " << v.size() << std::endl;
    std::cout << "capacity: " << v.capacity() << std::endl;
}
```

affiche :

```
size: 4
capacity: 4
size: 5
capacity: 8
```

Vous voyez ici qu'après avoir ajouté un élément dans le `std::vector` avec `push_back` (le résultat aurait été identique en utilisant `resize` ou n'importe quelle fonction qui ajoute des éléments à une collection), le nombre d'éléments est passé de 4 à 5, comme vous pouvez vous y attendre. En revanche, la capacité est passée de 4 à 8.

La stratégie de gestion de la réserve (comment la réserve est augmentée) n'est pas définie dans la norme C++, chaque compilateur est libre d'utiliser la méthode qu'il souhaite. Par exemple, la bibliothèque standard

du compilateur GCC double la réserve à chaque fois qu'elle doit l'augmenter, jusqu'à une limite fixée. La bibliothèque standard fournie avec Visual Studio augmente la mémoire de 1.5 à chaque fois. (À vérifier)

Au contraire, lorsqu'un élément est retiré d'une collection (par exemple avec `pop_back` ou `resize`), la taille de la réserve n'est généralement pas modifiée.

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    std::cout << "capacity: " << v.capacity() << std::endl;
    v.push_back(5);
    std::cout << "capacity: " << v.capacity() << std::endl;
    v.resize(2);
    std::cout << "capacity: " << v.capacity() << std::endl;
}
```

affiche :

```
capacity: 4
capacity: 8
capacity: 8
```

Après avoir redimensionné le `std::vector`, celui-ci continue d'avoir une réserve de 8 éléments.

Il est possible de manipuler directement la taille de la réserve. La fonction membre `reserve` permet de garantir que la collection pourra contenir un nombre fixé d'éléments. Si la réserve actuelle est plus petite que ce que vous souhaitez réserver, elle sera augmentée en conséquence. Cependant, si elle est déjà de taille suffisante, elle ne sera pas diminuée.

Pour forcer la diminution de la taille de la réserve, vous devez utiliser la fonction `shrink_to_fit`, qui force la taille de la réserve à zéro (dit autrement, cela veut dire que la capacité devient égale au nombre

d'éléments, `shrink_to_fit` ne supprimant pas d'éléments).

main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    v.reserve(5);
    std::cout << "capacity: " << v.capacity() << std::endl;
    v.reserve(10);
    std::cout << "capacity: " << v.capacity() << std::endl;
    v.reserve(5);
    std::cout << "capacity: " << v.capacity() << std::endl;
    v.shrink_to_fit();
    std::cout << "capacity: " << v.capacity() << std::endl;
}
```

affiche :

```
capacity: 5
capacity: 10
capacity: 10
capacity: 4
```

Notes sur les performances

Même si les problèmes de performances sont volontairement mis de côté dans cette première partie du cours, il est intéressant d'avoir quelques notions.

En particulier sur la gestion de la mémoire, il y a trois choses importantes à connaître :

- allouer de la mémoire est très coûteux ;
- copier de la mémoire est coûteux ;
- avoir des éléments contigus en mémoire (comme dans `std::vector`) est généralement plus efficace.

Le premier point peut intervenir lorsque vous créez trop de variables inutiles (mais le compilateur peut dans certains cas optimiser automatiquement ce point). Il interviendra aussi lorsque vous changez la capacité d'une collection. L'idéal est de réserver exactement ce dont vous aurez besoin, mais il ne faut surtout pas sous-estimer cette réserve. Si vous changez (volontairement ou non) la taille de la réserve, cela sera plus coûteux en termes de performance que de réserver une seule fois une capacité un peu plus grande. Le pire étant bien sûr de ne pas réserver du tout.

Le second point généralement mieux compris lorsque les copies sont explicites, par exemple dans le cas des appels de fonctions (que vous verrez par la suite). Il est parfois plus difficile de repérer les copies automatiques, en particulier dans les manipulations de collections. Lorsqu'un `std::vector` a besoin d'augmenter sa réserve, il ne peut pas simplement allouer la mémoire pour les nouveaux éléments, mais pour l'ensemble des éléments (si par exemple, un `std::vector` contient 10 éléments et que vous augmentez la taille à 20, il ne peut pas simplement allouer 10 éléments, il est obligé d'allouer 20 nouveaux éléments). Cela implique qu'il doit copier les anciens éléments dans la nouvelle zone mémoire allouée, ce qui est très coûteux aussi.

```
std::vector<int> v { 1, 2, 3, 4, 5 };  
    // v alloue 5 éléments  
    // puis v copie les valeurs dans cette mémoire allouée  
v.resize(10);  
    // v alloue 10 éléments  
    // v copie les valeurs 1, 2, 3, 4, 5 dans la nouvelle zone  
    // mémoire  
    // v initialise les autres éléments avec 0
```

Le dernier point est plus complexe à comprendre, cela nécessite de comprendre le fonctionnement des ordinateurs, en particulier des caches mémoire. Sachez simplement qu'une mémoire contiguë comme dans `std::vector` sera généralement plus rapide qu'une mémoire non contiguë comme dans `std::list`.

Ajouter et supprimer des éléments

Pour l'ajout et la suppression d'éléments, il est nécessaire de faire la distinction entre les collections séquentielles (`std::vector`, `std::list`, etc.) et les collections associatives (`std::map`, `std::set`, etc.). En effet, dans le premier cas, les éléments ne sont pas ordonnés automatiquement par la collection, il est donc possible d'ajouter des éléments au début, à la fin ou n'importe où dans la collection. Dans le second cas, les éléments sont organisés automatiquement, vous ne pouvez pas choisir de placer un élément à un emplacement particulier de la collection.

Utiliser les collections comme des piles

Vous avez déjà utilisé les fonctions de base pour ajouter et supprimer des éléments dans une collection séquentielle. Les mots-clés sont les suivants :

- *emplace* (sur place) pour créer un élément ;
- *push* (pousser) pour ajouter un élément ;
- *pop* (retirer) pour supprimer un élément ;
- *front* (devant) correspond au début d'une collection ;
- *back* (derrière) correspond à la fin d'une collection.

Avec ces cinq mots-clés, vous avez donc les fonctions suivantes :

- `emplace_front` pour créer au début ;
- `emplace_back` pour créer à la fin ;
- `push_front` pour ajouter au début ;
- `push_back` pour ajouter à la fin ;
- `pop_front` pour retirer au début ;
- `pop_back` pour retirer à la fin.

La différence entre `emplace` et `push` est que la première va créer un élément directement dans la collection, alors que la seconde va ajouter dans la collection un élément existant. Vous avez vu que certains types

sont déplaçables (*movable*), utiliser `push` sur ce type d'élément ne sera généralement pas coûteux. Mais lorsque c'est possible, il sera toujours plus performant d'utiliser directement `emplace`.

Insérer un élément

Comme pour les fonctions servant à ajouter un élément, les fonctions pour insérer des éléments sont déclinées en plusieurs versions. La fonction `insert` permet d'insérer un élément existant n'importe où dans une collection, tandis que la fonction `emplace` permet de créer directement un élément n'importe où dans une collection.

Dans une collection séquentielle

Dans le cas d'une collection séquentielle, l'insertion aura la forme générale suivante, dans laquelle l'itérateur correspond à l'emplacement de l'insertion.

```
Itérateur = insert(Itérateur, valeurs...)
```

L'itérateur correspond à la position dans la collection à laquelle seront insérés les nouveaux éléments, les anciens éléments étant décalés après les nouveaux éléments. L'itérateur retourné par la fonction `insert` correspond au premier élément inséré.

main.cpp

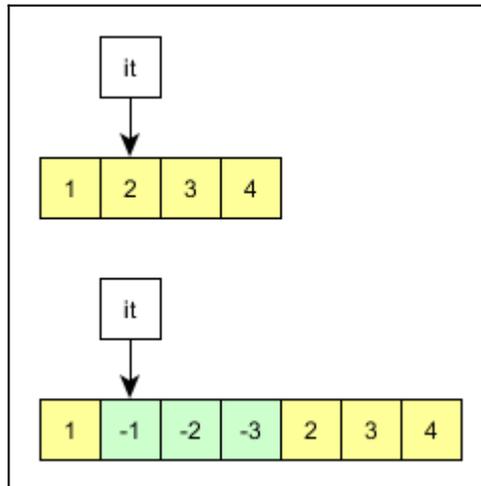
```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = std::find(begin(v), end(v), 2);
    v.insert(it, { -1, -2, -3 });
    for (auto i: v) std::cout << i << ' ';
}
```

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

affiche :

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



Il existe différentes versions de la fonction `insert`, permettant d'insérer une ou plusieurs valeurs.

```
auto it = v.insert(it, 123);           // insère la  
valeur 123  
auto it = v.insert(it, 3, 123);       // insère 3 fois  
la valeur 123  
auto it = v.insert(it, begin(w), end(w)); // insertion à  
partir d'une paire d'itérateurs  
auto it = v.insert(it, { 1, 2, 3 });  // insère la  
liste de valeurs 1, 2 et 3
```

La première version permet d'insérer un élément, la deuxième version permet d'insérer plusieurs fois la même valeur (faites bien attention à l'ordre des informations données : le nombre d'occurrence puis la valeur), la troisième version permet d'insérer plusieurs éléments provenant d'une collection (paire d'itérateurs), la dernière version permet d'insérer une

liste de valeurs.

Lorsque vous insérez des éléments dans un `std::vector`, les indirections sur les éléments peuvent être invalidés. Il n'est donc pas sûr d'insérer des éléments provenant d'un `std::vector` sur lui-même.

La fonction `emplace` n'existe qu'en une seule version.

```
auto it = v.emplace(it, 123); // insert la valeur 123
```

Dans un exemple aussi simple que le code précédent, il n'y aura pas de différence entre `insert` et `emplace`, mais cela sera important lorsque vous souhaitez insérer des éléments complexes.

Dans une collection associative

Dans une collection associative, les éléments sont organisés automatiquement, cela n'aurait pas de sens de vouloir insérer un élément à une position particulière. Dans la majorité des cas, vous insérerez directement une valeur dans une collection associative.

Cependant, il existe des versions de `insert` qui prennent un itérateur de position pour insérer un élément. La différence avec les collections séquentielles est que l'itérateur est seulement une suggestion, permettant à la collection d'être plus efficace pour insérer l'élément (il faut bien sûr que l'itérateur soit proche de la position finale à laquelle va être inséré l'élément. Si l'itérateur est quelconque, cela ne sera pas plus efficace).

Pour `emplace`, il existe également une version qui prend un itérateur de positionnement, mais la fonction change de nom. Il faut utiliser dans ce cas la fonction `emplace_hint`.

Les fonctions `insert`, `emplace` et `emplace_int` suivent les prototypes suivants :

```
// insert
```

```

std::pair<Itérateur, bool> = insert(valeur)
Itérateur = insert(Itérateur, valeur)
insert(valeurs...)

// emplace
std::pair<itérateur ,bool> = emplace(paramètres...)
Itérateur = emplace_hint(Itérateur, paramètres...)

```

Par exemple, pour insérer une liste de valeurs :

main.cpp

```

#include <iostream>
#include <set>

int main() {
    std::set<int> s { 1, 2, 3, 4 };
    s.insert({ -1, -2, -3 });
    for (const auto i: s) std::cout << i << ' '; std::cout
<< std::endl;
}

```

affiche :

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

Les différentes versions de `insert` et `emplace` sont similaires à celles des collections séquentielles.

```

// sans itérateur
s.insert(123); // insère la valeur 123
s.insert(begin(v), end(v)); // insertion à partir d'une
paire d'itérateurs
s.insert({ 1, 2, 3 }); // insère la liste de valeurs 1,
2 et 3

// avec itérateur
s.insert(it, 123); // insère la valeur 123

```

Selon la version des fonctions et le type de collection associative, les fonctions `insert`, `emplace` et `emplace_hint` peuvent retourner différentes informations :

- une paire contenant un itérateur sur l'élément inséré et un booléen indiquant si l'insertion a réussi ;
- un itérateur sur l'élément inséré ;
- rien du tout.

Lorsque l'insertion échoue, l'itérateur correspond à l'élément qui existe déjà avec la même clé.

Consultez la documentation pour connaître la syntaxe exacte de chaque fonction pour chaque type de collection. Par exemple, insérer un élément avec `insert` dans une `std::map` retourne une paire (l'insertion peut échouer, si `std::map` contient déjà un élément correspondante à la clé utilisée), alors que pour un `std::multimap`, cela retourne un itérateur (l'insertion ne peut pas échouer, il est possible d'avoir plusieurs éléments avec la même clé).

main.cpp

```
#include <iostream>
#include <map>

int main() {
    std::map<int, char> m { {1, 'a'}, {2, 'b'}, {3, 'c'} };
    const auto p = m.insert(std::make_pair(2, 'z'));
    for (const auto p: m)
        std::cout << p.first << ' ' << p.second << std::endl;
    std::cout << p.first->first << ' ' << p.first->second <<
    ' ' <<
        std::boolalpha << p.second << std::endl;
}
```

affiche :

```
1 a
2 b
3 c
2 b false
```

Suppression

Pour commencer, le plus simple : pour vider une collection (supprimer tous ses éléments), il suffit d'utiliser la fonction membre `clear`.

main.cpp

```
#include <vector>

int main() {
    vector<int> v { 1, 2, 3, 4 };
    v.clear();
}
```

Pour supprimer un ou plusieurs éléments, il faut utiliser la fonction `erase`. Celle-ci peut prendre un itérateur seul, pour supprimer un élément, ou une paire d'itérateurs, pour supprimer plusieurs éléments. Dans les deux cas, `erase` retourne un itérateur sur l'élément suivant le dernier élément supprimé (`erase` peut donc retourner `std::end`).

N'oubliez pas qu'en C++, une paire d'itérateurs correspond à un ensemble fermé à droite et ouvert à gauche (l'élément correspondant au premier itérateur est supprimé, l'élément correspondant au second itérateur n'est pas supprimé).

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

int main() {
    std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    auto first = std::find(begin(v), end(v), 3);
    auto last = std::find(begin(v), end(v), 7);
    v.erase(first, last);
    for (auto i: v) std::cout << i << ' '; std::cout << std::endl;
}
```

affiche :

```
1 2 7 8 9
```

Dans cet exemple, la fonction `erase` est appelée avec des itérateurs sur les éléments `3` et `7`. Après suppression, l'élément `3` a été supprimé, alors que l'élément `7` est encore dans la collection.

L'idiome `remove-erase`

Vous avez vu qu'insérer un élément dans `std::vector` impliquait que les éléments à la suite de l'élément inséré devaient être déplacés, pour laisser la place au nouvel élément. Pour la suppression, le problème est similaire : supprimer un élément dans un `std::vector` implique de déplacer tous les éléments à sa droite, pour combler l'emplacement qui a été libéré.

Lorsque vous souhaitez supprimer plusieurs éléments dans un `std::vector`, cela nécessitera alors de décaler plusieurs fois les éléments en fin de collection, ce qui peut être coûteux en termes de performances.

L'idiome “`remove-erase`” a un nom assez explicite : cela consiste à appliquer dans un premier temps l'algorithme `std::remove`, puis la fonction membre `erase`.

Le code d'exemple suivant utilise cet idiome sur une collection de caractères, qui contient des lettres et de chiffres, de façon à supprimer les chiffres.

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

int main() {
    std::vector<char> v { 'a', '1', 'b', '2', 'c', '3' };
    auto it = std::remove_if(begin(v), end(v), isdigit);
    v.erase(it, end(v));
    for (auto i: v) std::cout << i << ' '; std::cout << std
::endl;
}
```

affiche :

```
a b c
```

L'algorithme `std::remove` est un algorithme de la bibliothèque standard (il faut donc inclure le fichier d'en-tête `<algorithm>`). Il parcourt une collection et à chaque fois qu'il rencontre un élément à supprimer, il le remplace par l'élément valide suivant de la collection. A la fin, il retourne un itérateur sur le premier élément invalide de la collection (la taille de la collection n'a pas été modifiée).

Notez que "invalide" signifie que les éléments présent dans le collection sont encore accessible (les utiliser ne produit pas de comportement indéterminé), mais peuvent correspondre a n'importe quel élément. Le comportement est laisse a la discrétion des développeurs de la bibliothèque standard. Si vous avez la totalité de la collection du code précédent après l'appel a `std::remove`, vous pouvez obtenir par exemple le résultat suivant :

```
a b c 2 c 3
```

Vous voyez dans cette version de l'algorithme que les éléments valides sont places au début de la collection (comme attendu), les éléments invalides sont simplement laissés tel quel.

La fonction membre `erase` est ensuite utilisée pour supprimer les éléments invalides, qui sont les éléments entre l'itérateur retourné par `std::remove` et la fin de la collection.

En termes de performances par rapport a appeler plusieurs fois `erase`, cet idiome garantie que la complexité est linéaire, ce qui signifie que chaque élément n'est déplacé qu'une seule fois. (Vous verrez la complexité algorithmique au chapitre sur la création d'algorithmes).

Exercice : dans code précédent, il y a au total trois déplacements d'éléments avec l'idiome "remove-erase" (un pour chaque élément 'a', 'b' et 'c'). Essayez de calculer le nombre de déplacements qu'il faudrait si vous n'utilisez que la fonction `erase`.

Les idiomes de programmation

En programmation, un idiome est une écriture qui se retrouve habituellement dans un langage, pour réaliser une tâche particulière. Ils sont parfois peu compréhensible au premier abord pour un débutant, mais ils sont considérés comme faisant partie du “vocabulaire” commun des développeurs, qui faut savoir connaitre si on prêtant vouloir maîtriser un langage.

Notez bien que l'important n'est pas simplement de connaitre les idiomes, mais surtout de les étudier pour en comprendre les tenants et les aboutissants.

Accéder aux éléments

Premier et dernier éléments

Pour accéder aux éléments d'une collection, la méthode privilégiée est le parcours séquentiel, avec `std::begin` et `std::end`, qui fonctionne avec n'importe quel type de collection.

Certaines collections, comme `std::vector` et `std::list` proposent également des fonctions membres pour accéder directement au premier et au dernier élément, avec respectivement les fonctions `front` (*devant*) et `back` (*derrière*). Il faut bien sûr que la collection contienne au moins un élément, sinon cela provoque un comportement indéterminé.

main.cpp

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

int main() {
    std::vector<int> const v { 1, 2, 3, 4 };
    assert(!v.empty());
    std::cout << v.front() << std::endl;
    std::cout << v.back() << std::endl;
}
```

```
}
```

affiche :

```
1  
4
```

Notez que l'utilisation d'une assertion pour vérifier que le tableau n'est pas vide est inutile dans ce code, puisqu'il est déclaré juste au dessus, il serait difficile de faire une erreur. Mais c'est une bonne habitude à prendre de toujours faire les vérifications.

Accès aléatoire

Quelques types de collections, comme `std::vector` et `std::array` autorisent d'accéder directement à un élément quelconque, en utilisant les crochets droits `[POSITION]`. Le terme "aléatoire" signifie "n'importe quel élément" et pas "un élément au hasard. Les éléments sont identifiés par leur position dans la collection, en commençant avec l'indice 0.

Le type de l'indice est formellement donné selon le type de collection, en utilisant `size_type` sur la collection. Par exemple pour `std::vector` :

```
using type = std::vector<int>::size_type;
```

Ce type est un type entier non signé, qui correspond le plus souvent à `std::size_t` (qui représente une taille en général).

main.cpp

```
#include <iostream>  
#include <vector>  
#include <cassert>  
  
int main() {  
    const std::vector<int> v { 1, 2, 3, 4 };  
    const std::size_t index { 1 };  
    assert(index < v.size());  
    std::cout << v[index] << std::endl;
```

```
}
```

affiche :

```
2
```

Dans ce code, l'élément correspondant à `v[1]` n'est pas le premier élément de la collection, mais l'élément correspondant à l'indice "1", c'est à dire la valeur "2" (l'indice 0 correspondant au premier élément, c'est à dire la valeur "1").

(Le code d'exemple utilise volontairement des valeurs dans la collection proches des indices, pour insister sur le fait que vous ne devez pas penser en termes de "premier élément", "deuxième élément", etc. mais de "élément à la position 0", "élément à la position 1", etc.)

Comme toujours, l'accès en dehors du tableau provoque un comportement indéfini, il est donc nécessaire de tester la valeur de l'index avant d'accéder aux valeurs. L'écriture "index strictement inférieur à la taille" est une écriture idiomatique, il est préférable de l'utiliser.

Note : pour les collections n'autorisant pas un accès aléatoire, il est possible d'accéder à un élément en particulier, mais il faudra utiliser l'algorithme `std::advance` (qui parcourt une collection, en pratique).

main.cpp

```
#include <iostream>
#include <list>
#include <cassert>

int main() {
    std::list<int> const l { 1, 2, 3, 4 };
    const std::size_t index { 1 };
    assert(index < l.size());
    auto it = begin(l);
    std::advance(it, index);
    std::cout << (*it) << std::endl;
}
```

affiche :

```
2
```

Il existe également la fonction membre `at`, qui est similaire à l'opérateur `[]`, mais son usage n'est pas recommandé.

Collections associatives

Dans le cas des collections associatives, la notion de position n'a pas réellement de sens, puisque la collection est libre de réorganiser les éléments. Il est cependant possible d'accéder séquentiellement aux éléments, en utilisant l'algorithme `std::advance`, comme indiqué ci-dessus.

L'opérateur crochet droit `[]` n'est donc pas utilisable avec un indice dans la collection, mais avec une clé.

```
collection[clé]
```

Cet opérateur `[]` est utilisable aussi bien pour lire une valeur que pour la créer et la modifier. Si la clé n'existe pas, l'opérateur `[]` va créer un élément correspondant à cette clé.

`main.cpp`

```
#include <iostream>
#include <map>

int main() {
    std::map<int, char> m;
    std::cout << m.size() << std::endl;
    m[123] = 'a';
    std::cout << m.size() << std::endl;
    std::cout << m[123] << std::endl;
}
```

affiche :

```
0  
1  
a
```

La taille de la collection est bien augmenté de un après l'utilisation de l'opérateur `[]`.

Cela implique que l'accès en utilisant l'opérateur `[]` sur une collection associative est toujours valide, puisque cela va créer un élément si la clé n'existe pas. Vous pouvez quand même utiliser une assertion, dans le cas où vous souhaitez garantir qu'un élément ne sera pas ajouté dans la collection.

```
assert(m.find(key) == end(m));  
m[key] = value;
```

Il existe également une fonction membre `at`, qui est similaire à l'opérateur `[]`, mais son usage n'est pas non recommandé.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)