Les collections séquentielles

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

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

Sémantique de collection et des éléments

Vous avez déjà utilise a 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`).

En fait, 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 difficile a comprendre, cela fait partie de l'apprentissage du C++ d'apprendre a 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 a dire sans aucun parametre ou de liste de valeurs.

```cpp
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++.

```cpp
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.)

```cpp
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 ceux-ci directement lors de l'initialisation.

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

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

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

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

Il est également possible d'utiliser une liste de valeur 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 assignation (*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 copie dans la seconde collection. Au final, 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 retires de la première collection pour être déplacer dans la seconde. Au final, la première collection sera vide et la seconde contiendra les éléments qui se trouvaient précédemment dans la première collection.

main.cpp

```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);  // deplacement
    std::cout << v1.size() << ' ' << v3.size() << std::endl;
}
```

affiche :

```
3 3
0 3
```

Apres la copie, les collections `v1` et `v2` contiennent tous les deux trois éléments. Apres 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.

Etat d'un objet apres deplacement

Dans l'exemple precedent, le tableau `v1` ne contient plus d'elements apres le deplancement. En fait, en toute rigueur, ca n'est pas forcement vrai. Apres un deplacement, la norme C++ requiere qu'un objet soit valide, mais il peut etre dans n'importe quel etat. Il peut donc etre vide ou ne pas etre modifie.

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 copiée et déplacée par affectation.

main.cpp

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

int main() {
    const 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 a 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 a 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-a-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 ajoutes lors de la création en 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 supprimées.

La fonction membre `assign` est équivalente a une assignation 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 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), un instance particulière, par exemple `T<A>`, ne sera généralement pas convertible en une autre instance `T<B>`.

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 redimensionner 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 ne sont disponible uniquement 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 ajouter 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 `capactity` (capacité) permet de connaître la taille totale réservée par une collection.

main.cpp

```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 augment la mémoire de 1,5 à chaque fois. (A 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

```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 contiguës 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 changer la capacité d'une collection. L'idéal est de réserver exactement ce dont vous aurez besoin, mais il ne faut surtout pas sous estimé cette réserve. Si vous changer (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 sur de ne pas réserver du tout.

Le second point est plus généralement bien compris lorsque les copies sont explicites, par exemple dans la 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 augmenter 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 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
mémoire
  // v initialise les autres éléments avec 0
```

Le dernier point est plus complexe à comprendre, cela nécessite de comprendre la 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 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, a la fin ou n'importe ou dans la collection. Dans le second cas, les éléments sont organisés automatiquement, vous ne pouvez pas choisir de placer un élément a un emplacement particulier de la collection.

Ajouter un élément

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 quatre 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 second va ajouter dans la collection un élément existant. Vous avez vu que certain type

sont déplaçable (*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`.

Insertion

ou emplace

Collection séquentielle

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

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

L'itérateur correspond a la position dans la collection a laquelle sera insérer les nouveaux éléments, les anciens éléments étant décalées après les nouveaux éléments. L'itérateur retourné par la fonction `insert` correspond au premier élément inséré.

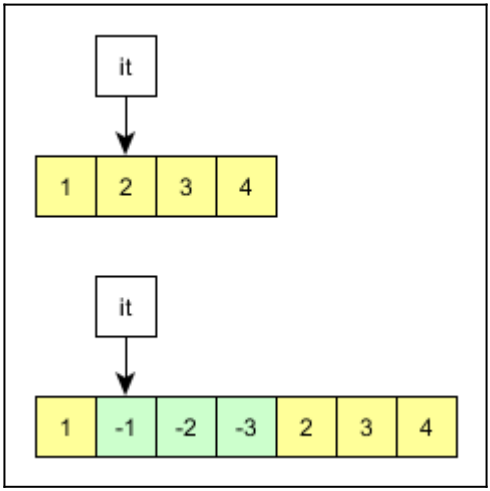
main.cpp

```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.

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

Collection associative

Dans le cas des collections associatives, la fonction `insert` existe en version avec ou sans itérateur. La différence avec les collections séquentielles est que l'itérateur est seulement une suggestion, permettant a la collection d'être plus efficace pour insérer l'élément (il faut bien sur que l'itérateur soit proche de la position finale a laquelle va être insérer l'élément. Si l'itérateur est quelconque, cela ne sera pas plus

efficace).

```
std::pair<Itérateur, bool> = insert(valeur)
Itérateur = insert(Itérateur, valeur)
insert(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 version de `insert` sont similaires a celles des collections séquentielles.

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

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

Selon la version, la fonction `insert` retourne différentes informations :

- lorsque vous inserer une valeur, `insert` retourne une paire contenant un iterateur et un booleen indiquant si l'insertion a reussit ;
- lorsque vous inserer une valeur en utilisant un iterateur, `insert`

retourne un iterateur ;
- lorsque vous inserer plusieurs valeurs (en utilisant une paire d'iterateurs ou une liste de valeurs), `insert` ne retourne rien.

multimap/multiset : insert ne peut pas echouer, pas de bool, retourne toujours sur l'element inserer. map/set : peut echouer, retourne bool. Iterateur sur nouvelle element ou sur element bloquant l'insertion.

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

L'idiome remove-erase

Accéder aux éléments

Accès aux éléments (random access, front, back)

Les autres fonctions membres

Autres fonctions membre (find, count, etc)

Itérateurs

allocator, data()

Chapitre précédent	Sommaire principal	Chapitre suivant