Les catégories de collections

La bibliothèque standard propose de nombreux conteneurs de données. Pour le moment, vous avez vu principalement `std::vector` et `std::string`. Ce chapitre présente les autres collections de la bibliothèque standard et leurs fonctionnalités de base.

Note : la manipulation plus poussées des collections avec `begin` et `end` passe par l'utilisation des itérateurs. Ce concept est suffisamment important pour faire l'objet d'un chapitre dédié. Ce chapitre se focalise uniquement sur les fonctionnalités spécifiques des conteneurs.

Collection et conteneur
En toute rigueur, les concepts de collection et conteneur sont différents. Le premier correspond à un objet qui propose les concepts de “premier élément”, “dernier élément” et “élément suivant”, le second correspond à un objet qui peut contenir d'autres objets.

En pratique, les conteneurs de la bibliothèque standard du C++ sont aussi des collections, vous verrez souvent dans ce cours les termes “collection” et “conteneur” utilisés indifféremment. (Lorsque vous créerez vos propres conteneurs, il est recommandé de les implémenter également sous forme de collections, pour être compatibles avec les algorithmes de la bibliothèque standard).

Mais n'oubliez pas que ce sont des concepts distincts (par exemple, les vues *views* proposent les fonctionnalités des collection, mais ne contiennent les données qu'elles manipulent. Et les graphes peuvent être des conteneurs de données, mais ne proposent généralement pas les concepts de “premier élément”, “dernier élément” et “élément suivant”).

Les différents types de conteneurs

Conteneurs séquentiels et associatifs

La première grande classification des collections est la séparation entre conteneurs séquentiels et associatifs. Un conteneur associatif associe une clé à des informations (un objet). A partir de cette clé, il est alors facile de retrouver les informations associées. C'est une notion que vous pouvez retrouver dans la vie de tous les jours : par exemple, pour trouver une définition (l'information) d'un mot (la clé) dans un dictionnaire (la collection de données), ou lorsque vous allez à la banque consulter vos dépenses du mois (les informations) en utilisant votre numéro de compte (la clé).

Un conteneur associatif va gérer en interne l'organisation des données pour optimiser les accès à partir de la clé. Il existe plusieurs types de conteneurs associatifs, qui se distinguent selon leur façon d'organiser les données et donc les facilités d'accès.

Au contraire, un conteneur séquentiel ne présente pas d'accès privilégié aux données selon une clé et n'organise pas les données en interne. Il est possible d'accéder aux données uniquement de façon séquentielle, c'est-à-dire en utilisant les concepts communs à toutes les collections : "début d'une collection", "fin d'une collection" et "élément suivant". Certains conteneurs séquentiels proposent des fonctionnalités supplémentaires, comme par exemple "accéder directement à l'élément n" ou "élément précédent".

Notez bien que les conteneurs associatifs sont aussi des collections et peuvent donc être manipulés comme des conteneurs séquentiels. Cependant, n'oubliez pas qu'ils gèrent automatiquement en interne les données, cela n'a pas de sens par exemple de trier les éléments avec `std::sort`.

Pour bien comprendre ce principe de "gestion automatiquement interne" des données par une conteneur associatif, le code suivant utilise une

conteneur non-associatif (`std::vector`) et un conteneur associatif (`std::map`) et affiche les éléments séquentiellement (avec une boucle `for` que vous verrez dans la site de ce cours).

main.cpp

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

int main() {
    using data_t = std::pair<int, std::string>;
    std::vector<data_t> v {
        { 2, "hello" },
        { 3, "every" },
        { 1, "body" }
    };
    for (auto & p: v)
        std::cout << p.first << ' ' << p.second << std::endl;

    std::cout << std::endl;

    std::map<int, std::string> m {
        { 2, "hello" },
        { 3, "every" },
        { 1, "body" }
    };
    for (auto & p: m)
        std::cout << p.first << ' ' << p.second << std::endl;
}
```

affiche :

```
2 hello
3 every
1 body

1 body
2 hello
3 every
```

Dans le conteneur non-associatif (`std::vector`), les éléments sont

affichés dans le même ordre qu'ils ont été entrés dans la collection. Dans le conteneur associatif (`std::map`), les éléments ont été trié en fonction de la clé (la valeur entière).

Vous pouvez voir dans le code précédent qu'il est possible de manipuler les mêmes informations, quelque soit le type de conteneur. Il est tout à fait possible de trier un tableau `std::vector` en fonction de la valeur entière avec `std::sort` et de faire une recherche sur cette valeur avec `std::find`. La différence est que dans la cas des conteneurs associatifs, l'une des informations a un rôle particulier pour le stockage et l'accès.

Choisir la structure de données la plus adaptée à une problématique fait partie des bases de la programmation.

Les tableaux

Le premier type de conteneur séquentiel (et probablement le plus utilisé) est le tableau. Vous connaissez déjà `std::vector` (tableau de taille dynamique) et `std::array` (tableau de taille fixée à la compilation), il existe également `std::valarray` (un tableau spécialisé pour les calculs arithmétiques, ce type est peu utilisé) et `std::dynarray` (un tableau de taille fixé à l'exécution, qui sera ajouté dans le C++17, mais qui est déjà disponible dans certains compilateurs dans `std::experimental::dynarray`).

Anciennes syntaxes

Il existe également deux types de tableaux hérités du C, de taille fixée à la compilation et de taille dynamique. La syntaxe est la suivante :

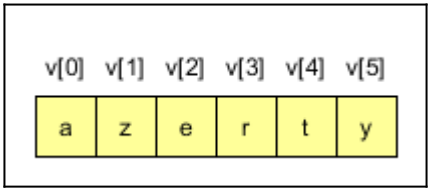
```
// taille fixe
int static_array[10];

// taille dynamique
int dynamic_array[] = new[10] int;
// utilisation...
```

```
delete[] dynamic_array;
```

L'utilisation correcte de ce type de tableau (et plus généralement la gestion manuelle de la mémoire avec `new` et `delete`) est relativement complexe et doit être évitée en C++ moderne. Les cas d'utilisation acceptables peuvent être la compatibilité avec un code C++ ancien ou avec le C, ou pour des implémentations de conteneurs, à partir du moment où ce type de code est correctement isolé du reste du programme (encapsulation, cela sera détaillé dans la partie sur la programmation orientée objet).

La particularité des tableaux est d'avoir ses éléments contiguës en mémoire, dans un bloc mémoire réservé pour ce tableau. Cela permet d'avoir un accès efficace direct à n'importe quel élément, selon son indice dans le tableau, avec l'opérateur `[]`. (Il est également possible d'utiliser la fonction `at`, mais son utilisation est déconseillée). Les indices commencent à partir de zéro jusque `size()-1`.

```
const vector<char> v { 'a', 'z', 'e', 'r', 't', 'y' };
std:cout << v[0] << std::endl;
```



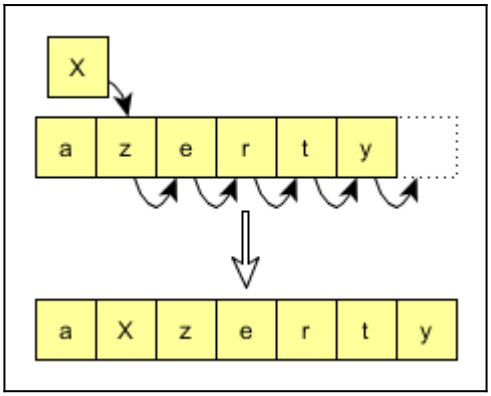
L'indice est une valeur entière positive, de type `vector<T>::size_type`, mais qui est en général similaire au type `size_t`.

L'accès en dehors des limites d'un tableau produit un comportement indéterminé. Il est de la responsabilité du développeur de vérifier les accès à un tableau, au minimum avec `assert`. Un indice est obligatoirement positif, mais il n'est pas nécessaire de vérifier cela si vous utilisez `size_t` (cela n'aurait pas de sens d'utiliser un type entier signé, comme `int`). Il faut également vérifier que la valeur ne soit pas plus grande que la taille du tableau.

```
const size_t i { ... };
assert(i < v.size());
v[i]; // ok
```

(Le code `i < v.size()` est strictement équivalent à `i ⇐ v.size()-1` pour les entiers, mais n'utilise qu'une seule opération de comparaison, alors que la seconde syntaxe utilise un comparaison et une soustraction, ce qui est moins performant).

Avoir les éléments contiguës en mémoire permet aux tableaux d'être très efficace (c'est plus optimisé pour les caches mémoires des ordinateurs modernes, mais cela sort du cadre de ce cours), mais impose également des contraintes. Par exemple, si vous souhaitez ajouter un élément au milieu du tableau du code précédent, il ne sera pas possible de placer cet élément directement dans le tableau (puisqu'il n'y a pas d'emplacement mémoire libre entre deux éléments). Il est alors nécessaire de déplacer tous les éléments qui se trouvent à droite de l'élément inséré, pour libérer un emplacement.



Il peut alors y avoir un nouveau problème. Cet insertion ne peut avoir lieu que s'il est possible d'utiliser l'emplacement mémoire situé directement à droite du dernier élément. Si ce n'est pas possible (en général parce qu'il y a déjà quelque chose à cet emplacement mémoire), il faudra alors créer un nouveau tableau en mémoire (avec la nouvelle taille), copier tous les éléments depuis l'ancien tableau vers le nouveau, puis insérer le nouvel élément. (Tout cela est réalisé automatiquement par `std:vector`

en interne, cela ne nécessite pas d'écrire un code spécifique de votre part).

Toutes ces copies et allocations de mémoire sont très coûteuses en termes de performances. De plus, cela invalide les indirections (cela sera détaillé dans un prochain chapitre). Il est donc important de limiter ce phénomène.

Pour cela, `std::vector` alloue en mémoire plus d'éléments que nécessaire. Lorsque vous insérer un nouvel élément, cette reserve sera utilisée en propriété et le tableau ne sera copié que si vous dépasser la capacité de cette reserve.

Cette réserve est en partie gérée automatiquement par `std::vector`, mais il est également possible de la gérer manuellement. Pour cela, vous pouvez utiliser les fonctions suivantes. Pour modifier la réserve :

- `reserve(n)` permet d'augmenter la taille de la réserve en mémoire. Cette fonction prend en paramètre le nombre d'éléments total que vous souhaitez. La réserve est augmentée uniquement si vous demandez plus que la réserve actuelle.
- `shrink_to_fit()` permet de réduire au maximum la taille de la réserve, de façon à ce que la taille totale correspond au nombre d'éléments réellement utilisés. (Autrement dit, la réserve sera mise à zéro).

Pour connaitre l'utilisation de la mémoire :

- `size()` permet de connaître le nombre total d'élément actuellement utilisés dans le tableau.
- `capactity()` permet de connaître le nombre totale d'élément dans le tableau (utilisés et réserve).
- `max_size()` permet de connaître le nombre maximal d'élément qu'il sera possible d'avoir dans un tableau.
- `empty()` permet de savoir si un tableau ne contient pas d'élément utilisé (autrement dit, si `size()` vaut zéro).

Pour avoir un code le plus performant possible, il faudra donc faire

attention de :

- créer les tableaux directement avec des valeurs (de préférence comme constant) ;
- réserver assez d'éléments pour éviter les ré-allocations de mémoire ;
- éviter les copies de tableaux.

```
// tableau constant
const std::array<int, 4> a { 1, 2, 3, 4 };

// tableau dynamique
std::vector<int> v { 1, 2, 3, 4 };
std::vector<int> v(100); // I

// réserve
std::vector<int> v;
v.reserve(100);
```

Notez bien la différence entre `std::vector<int> v(100)` et `v.reserve(100)`. La première syntaxe crée un tableau qui contient 100 éléments (la taille de la réserve est choisie automatiquement), la seconde crée un tableau vide (0 élément), mais qui contient 100 éléments dans la réserve.

Les listes

double et simple chaine

associatif simple ?

map, set,

unordored

Créer une collection

Construction (par défaut, par copie, reserve/resize). Size, capacité, empty, shrink_to_fit. swap

Ajouter et supprimer des éléments

Ajout et suppression d'éléments. L'idiome remove-erase. emplace vs push/insert

Complexité algo

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