

---

# PRINCIPES DES CONTENEURS ET ITERATEURS

---

# STL

**Introduction:** Jusqu'à présent, nous étions constamment obligés de recoder des algorithmes simples pour gérer des types de stockages comme des listes, files et autres conteneurs courants ou bien de récupérer des bibliothèques payantes ou pas mais en tout cas non standardisées. Les conteneurs de la STL viennent combler ce manque en apportant des méthodes de stockage rapides et efficaces.

**/!\ Ce document n'explique pas en détail les conteneurs mais les décrit afin d'en comprendre le concept, l'intérêt, les avantages et les inconvénients**

## VUE GENERALE DES CONTENEURS

Les conteneurs de la STL sont des méthodes de stockage basées sur les templates. Ce sont des récipients qui permettent de contenir n'importe quel type de base ou classe définie par l'utilisateur. Ces récipients sont accompagnés de toute une série de méthodes permettant de les manipuler, on peut ainsi généralement insérer, supprimer, remplacer avec une facilité déconcertante. On peut donc facilement contrôler le contenu et l'utilisation mémoire de ses variables. Les conteneurs standards se déclinent sous différentes formes pour créer au choix des tableaux, des listes chaînées, des piles, des files, des tableaux associatifs, etc. Aussi nombreux que soient les types de conteneurs, ils fonctionnent tous plus ou moins de la même manière, selon une interface de base commune, on stocke indifféremment dans un tableau ou dans une chaîne, la gestion des données étant laissée au conteneur. On peut cependant remarquer que certaines méthodes ne sont pas présentes pour tous les conteneurs pour une raison évidente de performance, on a donc à certains moments favorisé les performances par rapport à l'homogénéité mais cela n'empêche pas aux conteneurs de la STL d'offrir une liberté conséquente au développeur.

## LES ITERATEURS

Les itérateurs sont en quelque sorte des pointeurs sur un conteneur. On peut très facilement faire l'analogie entre le pointeur de tableau et l'itérateur. Il permet de voyager dans les éléments du conteneur sans se soucier ni du type d'objet stocké ni de la structure de fonctionnement interne du conteneur.

Avec tout conteneur de la STL est fourni deux types d'itérateurs :

**iterator**  
**const\_iterator**

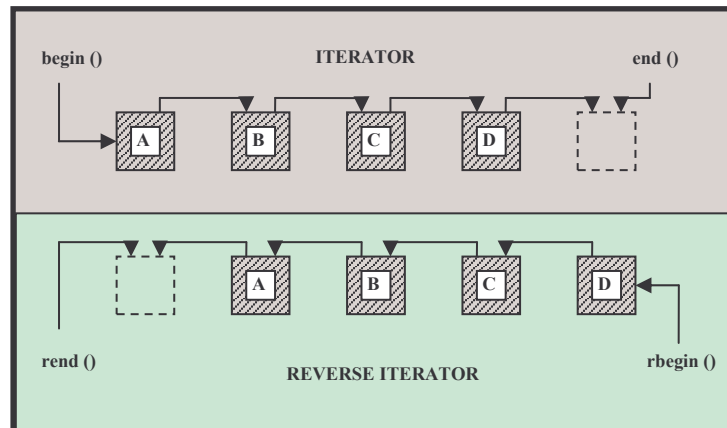
Le `const_iterator` a de particulier qu'il est uniquement destiné à la lecture.

Les itérateurs peuvent être classés dans 5 catégories différentes.

- **input\_iterator** : ce type d'itérateur est destiné à la consultation, il ne peut se déplacer que d'un élément à la fois et dans une seule direction.
- **output\_iterator** : ce type d'itérateur fonctionne comme le `input_iterator` mais est destiné à la modification.
- **forward\_iterator** : permet l'accès en écriture et en lecture mais ne peut toujours que se déplacer d'un élément à la fois et dans une unique direction.
- **bidirectional\_iterator** : permet de se déplacer dans les deux sens mais toujours d'un élément à la fois. C'est le type d'itérateur de `list`, on ne peut se déplacer que de maillon en maillon grâce à `it--` ou `it++`.

• **random\_access\_iterator** : permet de se déplacer d'un ou plusieurs éléments dans n'importe quelle direction pour lire ou écrire. Il permet d'utiliser ce genre de syntaxe : `it += 4` ou d'accéder à un élément par son index : `vect[4]` ; c'est le type d'itérateur du `vector`.

Il existe également le `reverse_iterator` qui comme son nom l'indique fonctionne en sens inverse. On peut récupérer des `reverse_iterator` grâce aux méthodes `rbegin()` et `rend()` des conteneurs.



## VECTOR

Le vecteur est le type de conteneur le plus proche du tableau classique. Il fonctionne de manière séquentielle et est stocké en mémoire sur une zone continue. Il peut s'utiliser comme un tableau avec néanmoins les avantages des méthodes et itérateurs (`random_access_iterator`). Il détecte automatiquement un dépassement mémoire et s'adapte en conséquence. Ce dernier point en fait son principal avantage comme son principal inconvénient. Lorsque sa taille se modifie, une réallocation de tout le tableau est opérée, si les réallocations sont trop fréquentes, elles peuvent amener à une chute de performance.

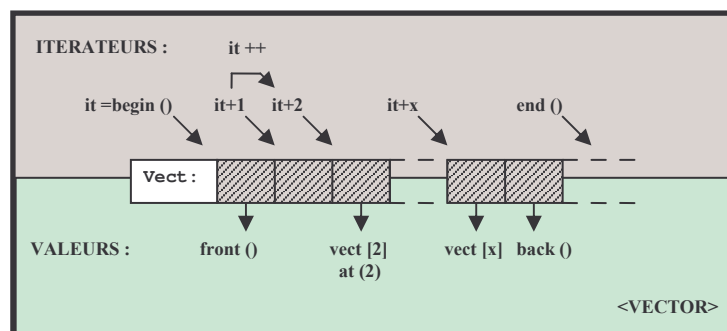


Schéma du type `vector`

Ce conteneur de part sa nature est adressable comme un tableau. Vu que l'on sait qu'il est alloué d'un bloc en mémoire, la navigation est aisée, on peut donc aller chercher ou modifier un élément en allant directement là où il est via `vect[ indice ]`, `at( indice )` ou en manipulant les itérateurs `it = it + 4`. Sur le schéma on peut voir la

distinction entre les fonctions manipulant des itérateurs (`begin`, `end`) et celles retournant des valeurs (`front`, `back`), il est important de faire la distinction, `begin` renvoie un itérateur sur le début de la chaîne et `front` renvoie le premier élément de la chaîne. `begin()+1` vous amènera à la deuxième position du vecteur et `front()+1` ajoutera 1 à la valeur retournée par la méthode, c'est-à-dire la première valeur du vecteur. On peut dans un vecteur insérer des valeurs assez facilement cependant cette opération n'est pas aussi coûteuse que dans le cas d'un ajout en fin de vecteur car dans tous les autres cas un décalage des données devra être opéré.

```
#include <iostream>
#include <vector> // entête pour utiliser vector
using namespace std;

int main () {
    typedef vector <int> vect;
    vect v;
    vect::iterator i;
    for (int a=0; a<10; a++) v.push_back(a); // on remplit
    for (i=v.begin(); i<v.end(); i++) // on affiche
        cout << " " << *i;
    cout << endl;
    system("pause");
    return 0;
}
```

Petit exemple d'utilisation du vector

## LIST

Le conteneur `list` est une liste doublement chaînée. Cette liste est donc composée de maillons contenant 2 pointeurs et la donnée. Ces maillons sont alloués dynamiquement et indépendamment à chaque fois que le besoin s'en fait sentir, on ne doit pas réallouer tout le contenu dans une zone continue de la mémoire, ce qui règle le problème du vecteur. Cependant l'inconvénient de cette méthode est que lorsque l'on doit accéder à une donnée, il faut parcourir la chaîne entière, on a donc un temps d'accès à la donnée qui peut être long. Si l'on doit aller chercher le dernier élément d'une chaîne de 10 000 éléments, on devra parcourir 9 999 pour rien. Par contre cette structure favorise les insertions en milieu de chaîne, les tris et les retournements de chaîne car il ne faut pour cela que modifier les pointeurs dans les maillons.

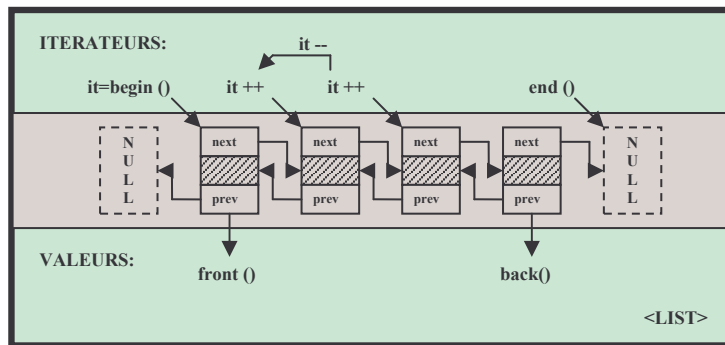


Schéma du type list

Vu la structure de la liste, on ne peut naviguer que d'un maillon à l'autre avec les itérateurs (bidirectional\_iterator) via `it--` ou `it++`. De plus l'accès direct est impossible, pas de fonction `at()` ou de surcharge de `[]`.

```
#include <iostream>
#include <list> // entête pour utiliser list
using namespace std;

int main () {
    typedef list <int> liste;
    liste l;
    l.push_back(3); // on insère 3 a la fin de la liste
    l.push_front(4); // on insère 4 au début de la liste
    for (i=l.begin(); i!=l.end(); i++) // on affiche
        cout << " " << *i;
    cout << endl;
    return 0;
}
```

Petit exemple d'utilisation de list

## DEQUE

Le conteneur deque est un conteneur qui prend les avantages de la liste pour les insertions en début et en fin de liste et ceux du vecteur pour ce qui est de l'accès aux éléments. On a un accès rapide pour les éléments du centre et on peut insérer efficacement et rapidement aux deux extrémités. Au final ça ressemble fort au vecteur à la différence que l'on peut facilement insérer au début, ce que le vecteur ne permet pas. Ce type de conteneur est particulièrement efficace pour des applications qui demandent d'ajouter ou supprimer des éléments aux extrémités, comme les files ou les piles. Avec ce type de conteneur on a une grande efficacité d'accès aux éléments, par contre l'efficacité de l'insertion au milieu est inversement proportionnelle à la distance aux extrémités.

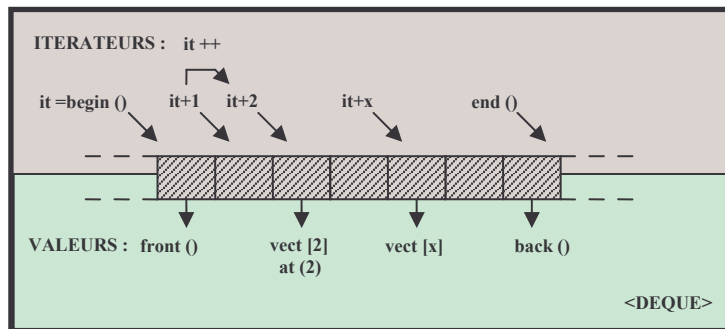


Schéma du type deque

Le parcours du deque se passe comme celui du vector.

```
#include <iostream>
#include <deque> // entête pour utiliser deque
using namespace std;

int main () {
    deque<int> int_d;
    int_d.push_back(2); // on insère 2 a la fin du deque
    int_d.push_front(1); // on insère 1 au début du deque
    for (int i=0; i<int_d.size(); ++i) // on affiche
        cout << int_d[i] << ' ';
    cout << endl;
    system("pause");
    return 0;
}
```

Petit exemple d'utilisation de deque

## STACK

Le conteneur stack est une pile (stack en anglais) donc fonctionnant sur base du système dernier entre, premier sorti. Il n'est pas un conteneur en lui-même, il se contente de se définir sur base d'un autre conteneur (deque par défaut). Il suffit au conteneur d'origine d'avoir les méthodes suivantes :

**back()** → récupérer la valeur du dernier élément  
**pop\_back()** → supprimer le dernier élément  
**push\_back()** → insérer un élément à la fin

qu'il renommera en **top()**, **pop()** et **push()**. On pourrait donc créer une pile sur base d'un vector mais le type deque est plus approprié.

```
#include <iostream>
#include <vector>
#include <stack> // entête pour utiliser stack
using namespace std;

int main () {
    typedef stack <int, vector<int> > s_i; // stack base sur un vector
    s_i stack;
    stack.push(3); // on insère 3
    stack.push(4); // on insère 4
    while (!stack.empty()) { // on affiche et on vide la pile
        cout << stack.top() << endl;
        stack.pop();
    } return 0;
}
```

Petit exemple d'utilisation de stack

## QUEUE

Le conteneur queue est une file donc fonctionnant sur base du système premier entré, premier sorti. Comme la pile (stack), ce n'est pas un conteneur en lui même, il se contente de redéfinir un autre conteneur (deque par défaut) contenant obligatoirement les méthodes :

**front()** → récupérer la valeur du premier élément  
**back()** → récupérer la valeur du dernier élément  
**push\_back()** → insérer un élément à la fin  
**pop\_front()** → supprimer le premier élément

qu'il renommara en **front()**, **back()**, **push()** et **pop()**.

## PRIORITY QUEUE

Le conteneur priority queue est comme son nom l'indique une file, cependant au lieu d'appliquer la méthode FIFO (first in, first out), la priority queue gère la priorité des éléments, plus la priorité d'un élément sera grande, plus vite il sortira. La fonction top() renverra donc l'élément avec la plus haute priorité. La priorité des éléments est déterminée par l'algorithme less<> par défaut qui utilise l'opérateur <, il faudra donc si l'on insère des classes home-mades, surcharger cet opérateur. On peut bien sur redéfinir l'ordre de priorité avec un autre algorithme de la STL ou un objet de comparaison créé pour l'occasion.

```

#include <iostream>
#include <queue> // entête pour utiliser priority_queue
using namespace std;

int main () {
    typedef priority_queue<int> pq_i; // priorite less par default
    pq_i queue;
    queue.push(3); // on insère 3, 40 et 4
    queue.push(40); // 40 est prioritaire, puis 4 et enfin 3
    queue.push(4);
    while (!queue.empty()) { // on affiche et on vide
        cout << queue.top() << endl; // affichera 40 4 3
        queue.pop();
    } return 0;
}

```

Petit exemple d'utilisation de `priority_queue`

## MAP

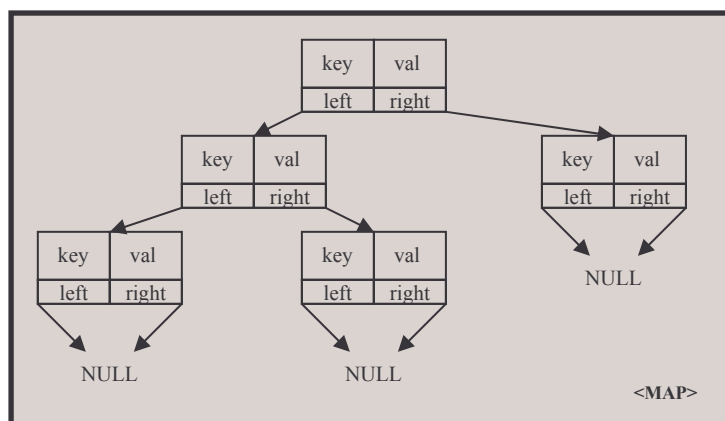


Schéma du type `map` (arbre)

Ce conteneur est un conteneur associatif, c'est-à-dire que les données insérées sont des paires. On peut par exemple créer un tableau associant le nom d'une personne (type `string`) et un numéro (`int`) et donc écrire : `tab["Onizuka"] = 1` ; Dans cet exemple on peut dissocier deux éléments : la clé et la valeur. Onizuka est la clé, qui va servir à ordonner les données dans la structure et 1 est la valeur qui lui est associée. Le conteneur `map` n'est pas organisé comme la liste ou le vecteur, les éléments ne se suivent pas, ils sont rangés dans un arbre afin de faciliter la recherche, la clé définissant leur position dans l'arbre. Il faut pour ce type de conteneur fournir un objet de comparaison ou lui laisser utiliser celui par défaut : `less<>` (qui a besoin de l'opérateur `<`) qui rangera les éléments dans l'ordre croissant. Il est également important de noter que pour un conteneur `map`, chaque clé doit être unique dans la structure. Les itérateurs de `map` sont des `bidirectional_iterator`, on peut donc faire `it++` et `it--` mais pas `it+1` et `it-1`.

```

#include <iostream>
#include <map> // entête pour utiliser map
using namespace std;

int main () {
    typedef map <string, int> tablo; // on laisse l'algo par défaut
    tablo :: iterator i; // on ne spécifie donc rien
    tablo tab; // d'autre que string et int
    tab["bla"]=10;
    tab["beuh"]=20;
    for (i=tab.begin(); i!=tab.end(); i++) // *i est une paire
        cout << (*i).first << " => " << (*i).second << endl; // first est la clé
    return 0 ; // second est la valeur
}

```

Petit exemple d'utilisation de map

## MULTIMAP

Le map associe une unique valeur à chaque clé, le multimap lui permet d'associer plusieurs valeurs à une même clé. La clé perd donc son unicité, on peut avoir deux fois la clé Onizuka avec des valeurs différentes. Par contre pour cette version, nous n'avons pas de surcharge de [], exit donc `tab["Onizuka"] = 1` ; on devra faire `tab.insert(pair<string,int>("Onizuka", 1))` ; . Ce conteneur est accompagné de toute une série de méthodes permettant de faire des recherches sur les clés (nombre de clés x, premier élément de clé x, dernier élément de clé x, etc.)

```

#include <iostream>
#include <map> // entête pour utiliser
using namespace std; // multimap (idem map)

int main () {
    typedef multimap<string, int> tablo;
    typedef pair<string, int> si_pair; // on laisse l'algo par défaut
    tablo :: iterator i; tablo tab; // on ne spécifie donc rien
    tab.insert(si_pair("beuh", 1)); // d'autre que string et int
    tab.insert(si_pair("beuh", 2));
    for (i=tab.begin(); i!=tab.end(); i++) // deux clés beuh
        cout << (*i).first << " => " << (*i).second << endl ;
    return 0 ;
}

```

Petit exemple d'utilisation de multimap

## SET / MULTISSET

Les conteneurs `set` et `multiset` fonctionnent comme `map` et `multimap`. La différence est qu'il n'y a pas de paire. On a juste une valeur qui joue également le rôle de clé. On pourra par exemple créer un arbre de `int` comme ceci : `set<int> arb` ; où les valeurs seront uniques. Si par exemple on voulait pouvoir insérer deux fois la valeur 2, on ferait `multiset<int> arb` ; . L'intérêt de ce type de conteneur est d'avoir une organisation dans laquelle il est facile de retrouver un élément. Il sera beaucoup plus rapide de chercher une valeur dans un `multiset` que dans un vecteur non trié (s'il est trié le vecteur a l'avantage). Dans le cas du vecteur, on cherchera sur la totalité de l'espace jusqu'à tomber sur l'élément tandis que dans le `multiset` on parcourra les branches en suivant la règle imposée à l'initialisation (par ex : si ce que je cherche est plus petit je vais voir dans la branche de droite, sinon celle de gauche), ce qui nous évitera une longue recherche sur tous les éléments. On a en quelque sorte un conteneur qui trie les éléments à leur insertion.

```
#include <iostream>
#include <set> // entête pour utiliser
using namespace std; // multiset(idem set)

int main () {
    typedef multiset<int> set_i; // on laisse l'algo par défaut
    set_i :: iterator i; set_i arb; // on ne spécifie donc rien
    arb.insert(1); // d'autre que int
    arb.insert(4); // deux clés/valeur 4
    arb.insert(4);
    arb.insert(2);
    for (i=arb.begin(); i!=arb.end(); i++) // affichera 1 2 4 4
        cout << *i << endl ;
    return 0 ;
}
```

Petit exemple d'utilisation de `multiset`