Les conteneurs

Les classes conteneurs

Les classes génériques permettent d'implémenter des collections capables de manipuler des types quelconques.

C++ fournit des classes génériques permettant d'implémenter toute sorte de collections dans la Standard Template Library (STL):

- Vecteurs (tableaux) ordonnés ou non : vector<T>
- Listes (listes chainées) : list<T> ou deque<T>
- Piles (LIFO) : stack<T>
- Files (FIFO) et files prioritaires: queue<T> et priority_queue<T>
- Conteneurs associatifs ordonnés : map<C,V> et multimap<C,T>
- Conteneurs associatifs non ordonnés : unordered_map<C,V> et unordered_multimap<C,T>
- Ensembles ordonnés : set et multiset
- Ensembles non ordonnés : unordered_set et unordered_multiset

Les classes conteneurs

•Les classes conteneurs de la STL sont conçues pour stocker toutes sortes d'objets.

•De nombreuses méthodes liées à ce type de classes imposent aux éléments de fournir les éléments suivants :

- •Un constructeur par défaut.
- •Un constructeur de copie.
- •Un opérateur d'affectation = .

Classe	Fichiers en-tête
vector	<vector>
list	<list>
deque	<deque>
stack	<stack>
queue	<queue>
priority_queue	<queue>
map	<map>
multimap	<map>
unordered_map	<unordered_map>
Unordered_multimap	<unordered_map>
set	<set>
multiset	<set>
unordered_set	<unordered_set>
unordered_multiset	<unordered_set>

Types de conteneurs

La bibliothèque classifie les conteneurs en deux grandes catégories :

- **<u>Les séquences :</u>**
  - Une séquence est un conteneur capable de stocker ses éléments de manière séquentielle, les uns à la suite des autres.
  - Les éléments sont identifiés par leur position dans la séquence.
  - L'ordre relatif est pertinent.
- **<u>Les conteneurs associatifs</u>**
  - Les conteneurs associatifs manipulent leurs données au moyen de valeurs qui les identifient indirectement.
  - Ces identifiants sont appelées des clés par analogie avec la terminologie utilisée dans les bases de données.
  - L'ordre relatif des éléments est géré selon la stratégie du le conteneur
  - La recherche des éléments se fait, généralement, par l'intermédiaire de leurs clés.

- Le choix des conteneurs de la bibliothèque standard doit s'effectuer en fonction de l'utilisation dans l'application ciblée.

Les itérateurs

- Un itérateur:
  - est un objet permettant d'accéder à tous les objets d'un conteneur donné, selon une interface standardisée.
  - ne représente pas les objets eux mêmes, mais constitue le moyen de les atteindre.
  - constitue une abstraction de la notion de pointeur pour les tableaux
  - utilise la même sémantique que les pointeurs et permet donc de parcourir tous les éléments d'un conteneur séquentiellement à l'aide de l'opérateur de déréférencement * et de l'opérateur d'incrémentation ++.
- tous les conteneurs de la bibliothèque standard disposent d'itérateurs.
- Les algorithmes manipulant les conteneurs sont donc écrits de manière uniforme indépendante de l'implémentation du conteneur.
- ce sont les conteneurs qui fournissent les itérateurs qui leur sont appropriés afin de permettre l'accès à leurs données.
- Il n'est donc pas nécessaire de comprendre le mécanisme interne du conteneur pour pouvoir parcourir ses éléments.

Utilisation des itérateurs

- Afin de permettre l'initialisation de leurs itérateurs, les conteneurs fournissent deux méthodes begin et end, qui renvoient respectivement un itérateur référençant le premier élément du conteneur et la valeur de fin de l'itérateur.
- Ainsi, le parcours d'une instance d'un conteneur de type *Conteneur* se fait de la manière suivante :

```
// Obtient un itérateur sur le premier élément
Conteneur::iterator i = instance.begin();
// Boucle sur toutes les valeurs de l'itérateur
while (i != instance.end()){
    // manipulation de l'élément référencé par i avec la
      //fonction f
    f(*i);
    // Passage à l'élément suivant
    ++i;
}
```

Les séquences

Conteneurs de type séquences

Type	Représentation
array : tableau statique (taille fixe)	
vector : tableau dynamique (taille dynamique)	
deque : double-ended queue (file permettant l'insertion au début et à la fin).	
list : liste doublement chainée	
forward_list : liste simplement chainée	

Fonctions membres des séquences

Headers		<array>	<vector>	<deque>	<forward_list>	<list>
Members		array	vector	deque	forward_list	list
iterators	begin	begin	begin	begin	begin before_begin	begin
	end	end	end	end	end	end
	rbegin	rbegin	rbegin	rbegin		rbegin
	rend	rend	rend	rend		rend
const iterators	cbegin	cbegin	cbegin	cbegin	cbegin cbefore_begin	cbegin
	cend	cend	cend	cend	cend	cend
	crbegin	crbegin	crbegin	crbegin		crbegin
	crend	crend	crend	crend		crend
capacity	size	size	size	size		size
	max_size	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty	empty
	resize		resize	resize	resize	resize
	shrink_to_fit		shrink_to_fit	shrink_to_fit		
	capacity		capacity			
	reserve		reserve			

C++98	Available since C++98
C++11	New in C++11

Source : http://www.cplusplus.com/reference/stl/

Fonctions membres des séquences

Headers		<array>	<vector>	<deque>	<forward_list>	<list>
Members		array	vector	deque	forward_list	list
access	front	front	front	front	front	front
	back	back	back	back		back
	operator[]	operator[]	operator[]	operator[]		
	at	at	at	at		
modifiers	assign		assign	assign	assign	assign
	emplace		emplace	emplace	emplace_after	emplace
	insert		insert	insert	insert_after	insert
	erase		erase	erase	erase_after	erase
	emplace_back		emplace_back	emplace_back		emplace_back
	push_back		push_back	push_back		push_back
	pop_back		pop_back	pop_back		pop_back
	emplace_front			emplace_front	emplace_front	emplace_front
	push_front			push_front	push_front	push_front
	pop_front			pop_front	pop_front	pop_front
	clear		clear	clear	clear	clear
	swap	swap	swap	swap	swap	swap

C++98	Available since C++98
C++11	New in C++11

Source : http://www.cplusplus.com/reference/stl/

Construction et initialisation

```
#include <iostream>
#include <list>
using namespace std;

void afficher (list<int>& l){
list<int>::iterator i = l.begin();
while (i != l.end()){
     cout << *i << " " ;
     ++i;
}
cout << endl;
}
```

```
// Initialise une liste avec trois
//éléments valant 5 :
list<int> l1(3, 5);
// Initialise une autre liste à partir de
//la première :
list<int> l2(l1.begin(), l1.end());
// Affecte 4 éléments valant 2 à l1 :
l1.assign(4, 2);
// Affecte l1 à l2
l2.assign(l1.begin(), l1.end());
```

Insertion et accès

Pour insérer des éléments dans une séquence, nous disposons de ces 3 surcharges de la méthode insert:

- **`iterator insert(iterator i, value_type valeur)`**
  Permet d'insérer une copie de la valeur spécifiée en deuxième paramètre dans le conteneur. L'insertion se fait immédiatement avant l'élément référencé par cet itérateur. Cette méthode renvoie un itérateur sur le dernier élément inséré dans la séquence.
- **`void insert(iterator i, size_type n, value_type valeur)`**
  Permet d'insérer n copies de l'élément spécifié en troisième paramètre avant l'élément référencé par l'itérateur i donné en premier paramètre.
- **`void insert(iterator i, iterator premier, iterator dernier)`**
  Permet d'insérer tous les éléments de l'intervalle défini par les itérateurs premier et dernier avant l'élément référencé par l'itérateur i.

En fonction du type de la séquence, nous disposons d'autre méthodes d'insertion et d'accès

- **`void push_front (const value_type& valeur)`**
  Ajoute la valeur en tête.
- **`void push_back (const value_type& valeur)`**
  Ajoute la valeur à la fin.
- **`value_type& front ()`:** retourne la valeur en tête.
- **`value_type& back ()` :** retourne la valeur à la fin.
- **`value_type& at (index)` :** retourne la valeur à la position index.
- **`value_type& Operator[] (index)` :** retourne la valeur à la position index.

Insertion et accès

Exemples :

```
list<int>::iterator i = l1.insert(l1.begin(), 5); // Ajoute 5 à la liste
l1.insert(i, 2, 3); // Ajoute deux 3 à la liste
list<int> l2;
l2.insert(l2.begin(),l1.begin(),l1.end());//Insère le contenu de l1 dans l2
l2.push_back(5);
cout<<l2.back(); //affiche 5
l2.push_front(6)
cout<<l2.front(); //affiche 6
```

Suppression

- La suppression dans une sequence s'effectue moyennant la méthode erase.
- la méthode erase a deux surcharges :
  - La première prend en paramètre un itérateur sur l'élément à supprimer.
  - La deuxième prend comme paramètres un couple d'itérateurs donnant l'intervalle des éléments de la séquence qui doivent être supprimés.
- Ces deux méthodes retournent un itérateur sur l'élément suivant le dernier qui a été supprimé ou l'itérateur de fin de séquence s'il n'existe pas de tel élément.
- La méthode clear() permet de vider le contenu de la séquence.
- Par exemple, la suppression du premier élément d'une liste peut être réalisée de la manière suivante :

```
list<int>::iterator i = instance.begin();
i = instance.erase(i);
```

- Selon le type de séquence, d'autres méthodes permettent de supprimer un element. Par exemples: `pop_front()` et `pop_back()`

Les listes

Le conteneur de type List

- implémente la structure d'une liste doublement chaînée d'éléments.
- implémente des itérateurs bidirectionnels

```
list<int> l1;
l1.push_back(2); //Ajout à la fin
l1.push_back(5);
cout << "Tête : " << l1.front() << endl; //affiche 2
cout << "Queue : " << l1.back() << endl; //affiche 5
l1.push_front(7); //Ajout en tête de liste
cout << "Tête : " << l1.front() << endl; //7
cout << "Queue : " << l1.back() << endl; //5
l1.pop_back(); //Suppression de la fin
cout << "Tête : " << l1.front() << endl; //7
cout << "Queue : " << l1.back() << endl; //2
```

- Les listes disposent également de méthodes spécifiques qui permettent de leur appliquer des traitements qui leur sont propres.
  - **remove(const T&)** : Permet de supprimer tous les éléments d'une liste dont la valeur est égale à la valeur passée en paramètre.
  - **Reverse** : Inverse l'ordre des éléments de la liste.

Les vecteurs

- La classe template vector de la bibliothèque standard fournit une structure de données dont la sémantique est proche de celle des tableaux.
- L'accès aux données est de manière aléatoire et donc réalisable en un coût constant.
- L'insertion et la suppression des éléments dans un vecteur ont des conséquences nettement plus lourdes que dans le cas des listes.
- Tout comme la classe list, la classe template vector dispose des méthodes **front** et **back** qui permettent d'accéder respectivement au premier et au dernier élément des vecteurs.
- Seules les méthodes **push_back** et **pop_back** sont définies.

Exemple :

```
vector<int> v(10); // Crée un vecteur de 10 éléments (des zéros)
v.at(2) = 2; // Modifie un élément
v.resize(11); // Redimensionne le vecteur
v.push_back(13);   // Ajoute un élément à la fin du vecteur
cout << v[5] << endl; // Affiche l'élément à l'indexe 5
```

Les adaptateurs de séquence

Les piles

- La classe template **stack** implémente la structure de données des piles.
- Une pile ne permet d'accéder qu'à l'élément situé en haut de celle-ci.
- La récupération des éléments se fait dans l'ordre inverse de leur empilement.
- La classe **stack** utilise deux paramètres template : le type des données lui-même et le type d'une classe de séquence implémentant au moins les méthodes **back**, **push_back** et **pop_back** (list, deque ou vector)
- Par défaut, la classe stack utilise deque comme structure et il n'est pas nécessaire de spécifier le type du conteneur à utiliser pour réaliser la pile.
- La classe stack propose les méthodes suivantes:
  - **push** permet d'empiler un élément sur la pile.
  - **pop** permet de retirer l'élément au sommet.
  - **top** : renvoie la valeur du sommet.

Les files

- La classe template **queue** implémente la structure de données des files.
- La récupération des éléments se fait dans le même ordre de leur insertion.
- La classe **queue** utilise deux paramètres templates : le type des données lui-même et le type d'une classe de séquence implémentant au moins les méthodes **front**, **back**, **push_back** et **pop_front** (list et deque seulement)
- Par défaut, la classe queue utilise une deque et il n'est pas nécessaire de spécifier le type du conteneur à utiliser pour réaliser la file.
- La classe deque propose les méthodes suivantes:
  - **front** et **back** : permettent respectivement de lire les valeurs du premier et du dernier éléments.
  - **push et pop :** permettent respectivement d'ajouter un élément à la fin de la file et de supprimer l'élément qui se trouve en tête de file

Les conteneurs associatifs

Les conteneurs associatifs

- les conteneurs associatifs sont capables d'identifier leurs éléments à l'aide des valeurs de leurs clefs.
- les conteneurs associatifs sont capables d'effectuer des recherches d'éléments de manière extrêmement performante.
- La bibliothèque standard distingue deux types de conteneurs associatifs :
  - les conteneurs qui différencient la valeur de la clef de la valeur de l'objet lui-même. Ces conteneurs représentent des associations clé-valeur (ayant le suffixe **map**).
  - les conteneurs qui considèrent que les objets sont leur propre clef. Ces conteneurs représentent des ensembles servant généralement à indiquer si un objet fait partie ou non d'un ensemble d'objets (ayant le suffixe **set**).
- Il existe des conteneurs associatifs à clé unique et d'autres à clé multiple (**multi**).
- Les associations à clefs uniques et à clefs multiple sont implémentées respectivement par les classes templates **map** et **multimap (c++11 unordered_map et unordered_multimap)**
- Les ensembles à clefs uniques et à clefs multiples sont implémentés respectivement par les classes templates **set** et **multiset (c++11 unordered_set et unordered_multiset)**.

Algorithmes

- La bibliothèque standard propose plusieurs algorithmes utilies applicables aux séquences et au conteneurs associatifs.
  - Mélange
  - Permutation
  - Rotation
  - Copie
  - Recherche
  - Ordonnancement
  - Comparaison personnalisée
  - Opérateurs ensemblistes

Autres ressources

- Visualisation:
  - https://www.freecodecamp.org/news/10-common-data-structures-explained-with-videos-exercises-aaff6c06fb2b/
  - https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
  - https://visualgo.net/en/list
  - https://visualgo.net/en/hashtable
  - https://visualgo.net/en/bst
  - https://visualgo.net/en/heap
- Complexité:
  - https://www.bigocheatsheet.com/
  - https://www.youtube.com/watch?v=__vX2sjlpXU