

# **Cours n° 6**

# **Standard Template Library I**



# Sommaire

---

## **1. Conteneurs séquentiels**

1.1 Séquences élémentaires

1.2 Adaptateurs de conteneurs

## **2. Itérateurs**

---

## **Standard Template Library (STL)**

Conception suivant les mécanismes de la généricité des langages impératifs,  
Normalisation en 1995 (ANSI Draft), 1997 (STL 3.0), 2003 (ISO/IEC 14882-2003)  
Source et documentation disponible sur [www.sgi.com/tech/stl/](http://www.sgi.com/tech/stl/)

### **Ensemble normalisé**

Patrons de classes des structures de données usuelles,  
Patrons de d'algorithmes classiques optimisées

### **Abstraction de l'implémentation**

Structures de données et algorithmes doivent fonctionner pour tout type d'élément  
    type intégré (au langage)  
    type utilisateur (défini par le programmeur utilisateur)  
Efficacité vitesse d'exécution

**B. Yablonsky, C++11 Standard Library: Usage and Implementation, CreateSpace , 2013.**

**S. Meyers, Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library, Addison-Wesley Professional, 2001.**

**A. Geron & F. Tawbi, Pour mieux développer avec C++ : Design patterns, STL, RTTI et smart pointers, Dunod, 2003.**

### **Sites**

**[en.cppreference.com/w/cpp/container](http://en.cppreference.com/w/cpp/container)**

**[www.sgi.com/tech/stl/table\\_of\\_contents.html](http://www.sgi.com/tech/stl/table_of_contents.html)**

## Composants de STL

### Conteneurs (container)

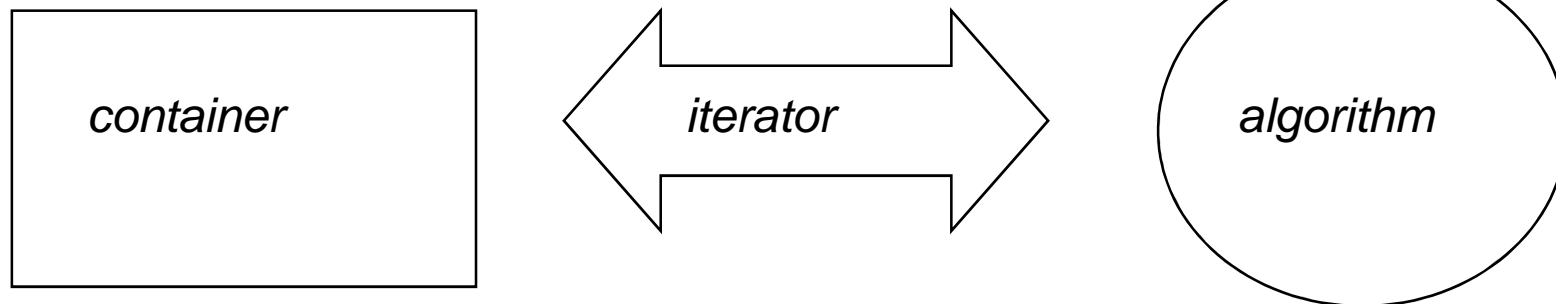
Implémentation des structures de données les plus courantes avec leur allocation/désallocation mémoire automatique

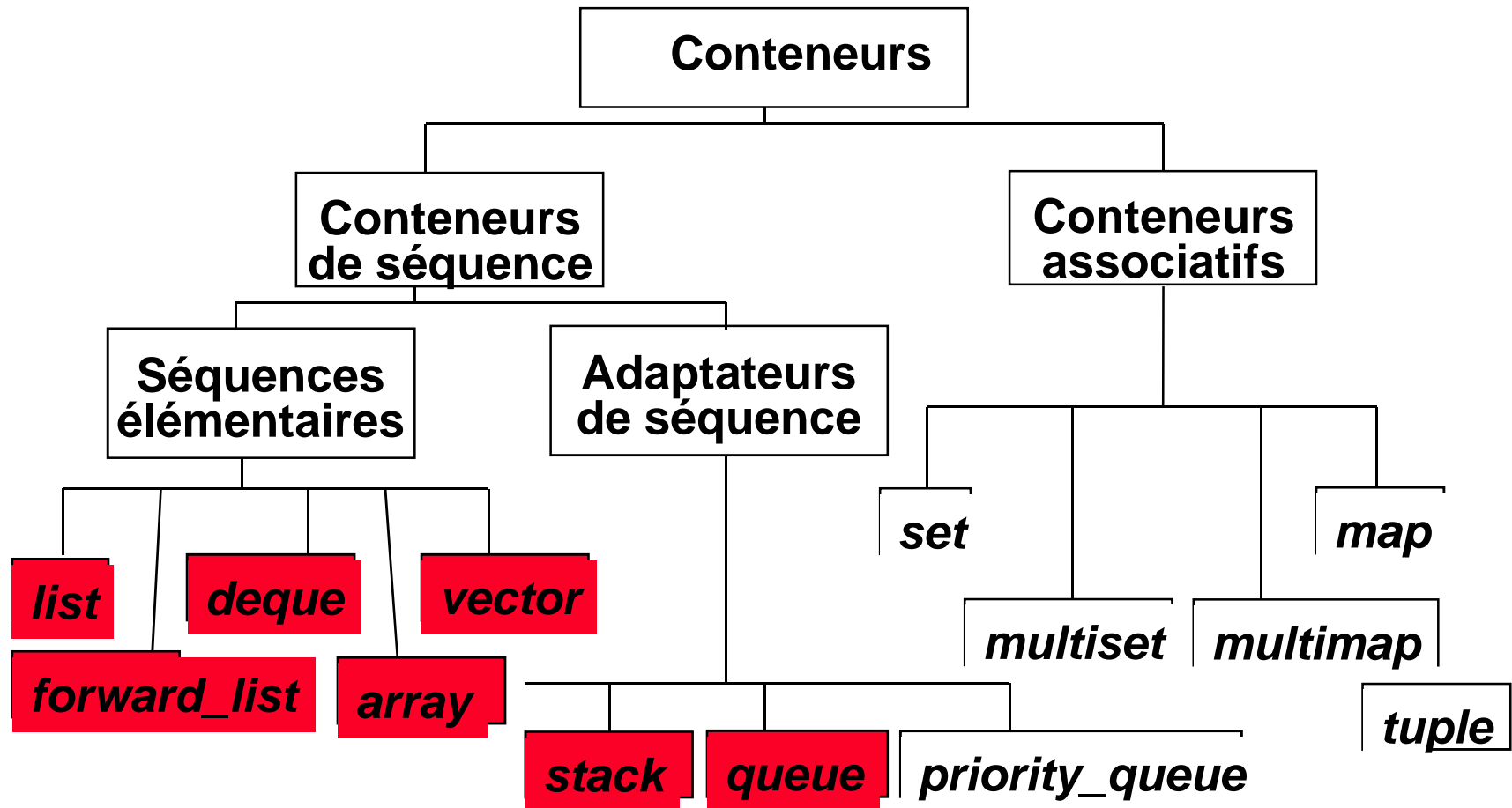
### Itérateurs (iterator)

Principes d'accès/balayage des éléments des conteneurs

### Algorithmes (algorithm)

Implémentation d'algorithmes génériques associés





### Patrons communs de fonctions membres

`bool empty();`

true si le conteneur est vide

`size_type size() const;`

Nombre d'éléments du conteneur

`T& front();`

Référence sur le premier élément

`T& back();`

Référence sur le dernier élément

`void push_back(const T& );`

Ajout (recopie) de l'élément en fin

`void swap(sequence <T>&back())`

Echange des éléments de this avec ceux du conteneur argument

### Instanciation, propriétés, et fonctions spécifiques

**array <T> v;** Déclaration d'un tableau de v d'éléments de type T

**array <T> v(n);** Déclaration d'un tableau de v de n d'éléments de type T

### Tableau générique sans insertion (sauf en fin)

Stockage dans un seul bloc contigu

**T& operator []**(size\_type); et **T& at**(size\_type);

Accès direct



## 1.2 SEQUENCES ELEMENTAIRES - CONTENEUR ARRAY

## Exemple

```
#include <iostream>
#include <array>
using namespace std;
int main() {
    int tabEnt[] = {10, 15, 23, 67, 43, 98, 34, 77};
    array<int> vi;
    for (int i = 0; i < sizeof(tabEnt) / sizeof(int); i++)
        vi.push_back(tabEnt[i]);

    cout << vi.size() << " ";
    for (int i = 0; i < vi.size(); i++) cout << vi[i] << " ";
}
```

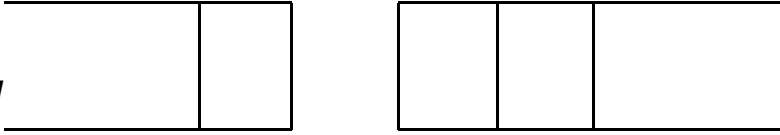
```
8  10 15 23 67 43 98 34 77
```

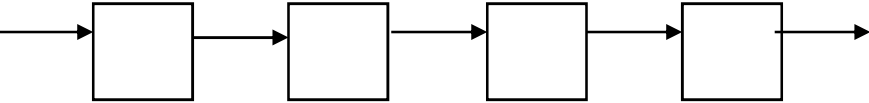
### Principes de fonctionnement

#### Blocs contigus d'objets (vector) ou non contigus (deque, list)

Insertion possible en tout point de la séquence,  
Performance fonction du type de séquence et du point d'insertion.

*vector*  Accès direct  
Insertion optimisée en fin

*deque*  
(Double End  
QUEue)  Accès direct  
Insertion optimisée  
en début ou en fin

*list*  Accès séquentiel  
Insertion et retrait  
optimisée en tout point

### Patrons communs de fonctions membres

void **resize**(size\_type, T c =T());

Redimensionnement du conteneur

iterator **insert**(iterator, const T& = T());

Insertion d'un élément à l'emplacement spécifié par l'itérateur

void **insert**(iterator position, iterator debut, iterator, fin)

Insertion à la position spécifiée les éléments de [debut fin[

void **pop\_back**();

Suppression de l'élément en fin de séquence

void **erase**(iterator i);

Suppression de l'élément pointé par l'itérateur i

void **erase**(iterator debut, iterator fin);

Supprime des éléments de [debut fin[

### Instanciation, propriétés, et fonctions spécifiques

**vector <T> v;**

Déclaration d'un vecteur v d'éléments de type T

**vector <T> v(n);**

Déclaration d'un vecteur v de n d'éléments de type T

### Extension de la structure classique de tableau (liste)

Création et variation de la taille au cours de l'exécution

Fonctions d'insertion et de suppression non optimisées (sauf en fin de liste)

size-type **capacity()** const;

Capacité actuelle du conteneur

void **reserve**(size\_type);

Ajout de capacité

T& **operator []**(size\_type); et T& **at**(size\_type);

Accès direct

## 1.2 SEQUENCES ELEMENTAIRES - CONTENEUR VECTOR

## Exemple

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int tabEnt[] = {10, 15, 23, 67, 43, 98, 34, 77};
    vector <int> vi;
    for (int i = 0; i < sizeof(tabEnt) / sizeof(int);i++)
        vi.push_back (tabEnt[i]);

    cout << vi.size() << " ";
    for (int i = 0; i < vi.size();i++) cout << vi[i] << " ";
}
```

```
8  10 15 23 67 43 98 34 77
```

### Instanciation, propriétés, et fonctions spécifiques

**deque <T> d;**

Déclaration d'une deque d d'éléments de type T

**deque <T> d(n);**

Déclaration d'une deque d n d'éléments de type T

### Fonctionnalités voisines de celles du conteneur vector

- + Insertion et Suppression en début de séquence plus rapide
- Accès direct plus lent

void **push\_front**(const T&) const;

Insertion d'un élément en début de séquence

void **pop\_front**();

Suppression en début de séquence

T& **operator []**(size\_type); et T& **at**(size\_type);

Accès direct

## 1.2 SEQUENCES ELEMENTAIRES - CONTENEUR DEQUE

## Exemple

```
#include <iostream>
#include <deque>
#include "Date.h"

using namespace std;

int main() {
    deque <Date> di;
    Date today, dfin;

    cin >> today; cin >> dfin;
    do { ++today; di.push_front(today); } while (today < dfin);

    cout << di.size() << " ";
    for (int i = 0; i < di.size(); i++) cout << di[i] << " ";
}
```

### Instanciation, propriétés, et fonctions spécifiques

**list <T> l;**

Déclaration d'une liste l d'éléments de type T

**list <T> l(n);**

Déclaration d'une liste l de n d'éléments de type T

### Spécialisation de la structure classique de liste

+ Fonctions rapides d'insertion et de suppression  
- Accès séquentiel,  
fonctions rapides (tri, concaténation, ...)

void **push\_front**(const T&) const;

Insertion d'un élément en début de séquence

void **pop\_front**();

Suppression en début de séquence

void **remove**(const& T valeur);

Suppression des éléments égaux à valeur

void **remove\_if**(Predicate pred);

Suppression des éléments vérifiant pred

void **sort**();

Tri des éléments selon l'opérateur <

void **reverse**();

Ordre inverse selon l'opérateur <

void **merge**(list<T> & x)

Fusion de la liste triée avec une autre liste x

void **unique**();

Suppression sur une liste triée des éléments en double

....



## 1.2 SEQUENCES ELEMENTAIRES - CONTENEUR LIST

## Exemple

```
int main() {  
    list<Date> ld;  
    Date fin, d;  
    cin >> d;  
    do {ld.insert(ld.end(), d); cin >> d; } while (d != fin);  
  
    cout << "liste de " << ld.size() << " éléments" << endl;  
    ld.sort();  
    cout << "entre " << ld.front();  
    ld.reverse();  
    cout << "et " << ld.front() << endl;  
}
```

```
liste de 3 éléments  
entre 19 6 2006  
et 21 6 2006
```

## Introduction

---

### Conteneurs implémentés à partir de séquences élémentaires

#### stack

Structure de données de pile - Last In First Out (LIFO)  
Insertions et retraits sur le haut de la pile

#### queue

Structure de données de file - First In First Out (FIFO)  
Insertions en fin et retraits en début

**Pas d'autres accès (direct ou séquentiel)**

### Instanciation, propriétés, et fonctions membres

**stack<T, vector<T> > p;**      Déclaration d'une pile d'éléments de type T

### Implémentation d'une structure abstraite de pile

#### Fonctions membres

bool **empty**();

    true si la pile est vide

size\_type **size**() const;

    Taille de la pile

T& **top**() :

    Lecture du sommet de la pile

void **push**(const T& );

    Empiler l'élément

T& **pop**()(const T& );

    Dépiler un élément

## 1.3 ADAPTATEUR DES SEQUENCES – ADAPTATEUR STACK

## Exemple

```
#include <vector>
#include <stack>
#include "../Cours 03/Animal.h"
int main(){
    stack <Animal, vector<Animal> > pa;

    pa.push(*(new Animal("lion", 4)));
    pa.push(*(new Animal("kangourou", 2)));
    pa.push(*(new Animal("araignée", 6)));

    while (pa.empty() == false) {pa.top().presente(); pa.pop(); }
}
```

```
je suis un représentant de l'espèce des araignée et j'ai 6 pattes
je suis un représentant de l'espèce des kangourou et j'ai 2 pattes
je suis un représentant de l'espèce des lion et j'ai 4 pattes
```

### Instanciation, propriétés, et fonctions membres

**queue<T, list<T> > f;**

Déclaration d'une file d'éléments de type T

### Implémentation d'une structure abstraite de file

#### Fonctions membres

bool **empty**();

true si la file est vide

size\_type **size**() const;

Taille de la file

T& **front**() :

Lecture de la tête de la file

void **push**(const T& );

Ajout de l'élément en queue

T& **pop**()(const T& );

Suppression d'un élément en tête

## 1.3 ADAPTATEUR DES SEQUENCES – ADAPTATEUR QUEUE

## Exemple

```
#include <iostream>
#include <list>
#include <queue>
using namespace std;
int main() {
    char tabChr[] = {'p', 'a', 'r', 'i', 's'};
    queue <char, list<char> > fa;
    for (int i = 0; i < sizeof(tabChr) / sizeof(char); i++)
        fa.push(tabChr[i]);
    while (fa.empty() == false) {
        cout << fa.front() << " "; fa.pop();
    }
}
```

p a r i s

## Variable qui repère la position d'un élément dans un conteneur

Utilisés pour accéder/balayer les éléments d'un conteneur

Généralisation et amélioration de la sûreté des pointeurs

## Quelque soit la classe conteneur : 2 méthodes membres

Début de la collection (position du premier élément)

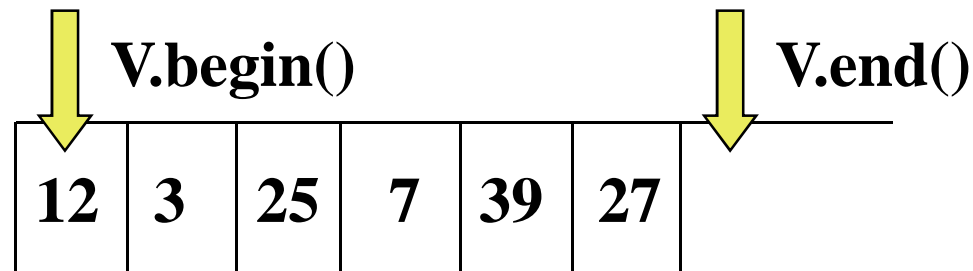
Itérateur renvoyé par la méthode membre `begin()`

Fin de la collection (position de l'élément "juste après le dernier")

Itérateur renvoyé par la méthode membre `end()`

Exemple :

```
vector<int> V;
```



### Opérateurs associés

#### Déréférencement

Accès au conteneur dont l'itérateur pointe l'élément `*It`

#### Déplacement (En avant, en arrière ou aléatoire)

Exemples :        `++It; It++;` // balayage séquentiel avant  
                  `--It; It--;` // balayage séquentiel arrière  
                  `It+=5; It-=3;` // accès direct

#### A chaque conteneur son type d'itérateur

Itérateur à accès aléatoire (vector et deque)

Itérateur bidirectionnel (list, map, set)

#### Types membres des classes conteneurs

`const_iterator` (version constante)

`iterator` (version non constante)

Exemples :        Déclaration d'un itérateur `Ivc` sur un vecteur de Complexe  
                  `vector<Complexe>::iterator Ivc;`



## 2. ITERATEURS

### Hiérarchie

#### *InputIterator*

Itérateur d'entrée  
Lecture  
Déplacement avant

{  
\*, i++, ++i  
Constructeur copie  
Opérateur d'assignation  
==, !=  
}

#### *OutputIterator*

Itérateur de sortie  
Ecriture  
Déplacement avant

#### *ForwardIterator*

Itérateur unidirectionnel  
Lecture et écriture  
Déplacement avant

#### *BidirectionalIterator*

Itérateur bidirectionnel  
Lecture et écriture  
Déplacement avant/arrière

i--, --i

#### *RandomAccessIterator*

Itérateur accès aléatoire  
Lecture et écriture  
Accès aléatoire

<, <=, >, >=  
+, +=, -, -=  
[]

## 2. ITERATEURS

## Balayage/accès par itérateur

```
typedef list<Date> listDate;  
typedef listDate::iterator itLDate;  
listDate L; itLDate it;  
  
// 1ère version  
for (it = L.begin(); it != L.end(); it++)  
    cout<< *it << " ";  
cout<<endl;  
  
// 2ème version  
it = L.end();  
do { --it; cout << *it << " "; }  
while (it != L.begin());  
  
return 0; }
```