# Chapitre 9 – Structures de données et STL

## 1. Introduction à la STL (Standard Template Library)

La **STL** est une partie essentielle de C++ moderne, introduite avec la norme **C++98** et enrichie dans les versions suivantes.
Elle est constituée de trois grands éléments :

| Élément | Description |
|---|---|
| **Conteneurs** | Structures de données génériques : vector, list, map, stack, queue, etc. |
| **Itérateurs** | Permettent de parcourir les conteneurs de manière uniforme. |
| **Algorithmes** | Tri, recherche, transformation, etc. (sort, find, count, …). |

La STL est **performante**, **générique**, **sécurisée** et **hautement optimisée**.

## 2. Le conteneur vector (vecteur dynamique)

std::vector ≈ un **tableau dynamique** :

- taille variable
- accès rapide en O(1)
- gestion automatique de la mémoire
- contigu en mémoire (compatible C)

Exemple :

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

int main() {
   vector<int> v = {1, 2, 3};
   v.push_back(4);
   for (int x : v)
      cout << x << " ";
}
```

Fonctions importantes :

| Fonction | Description |
|---|---|
| push_back(x) | ajoute à la fin |
| pop_back() | retire le dernier |
| size() | nombre d'éléments |
| capacity() | capacité mémoire |
| v[i] | accès direct |
| at(i) | accès sécurisé |
| clear() | vide le vecteur |

Complexité

| Opération | Complexité |
|---|---|
| Accès v[i] | O(1) |
| Insertion fin | amorti O(1) |
| Insertion au milieu | O(n) |

**3. Le conteneur list (liste doublement chaînée)**

std::list ≈ une **liste chaînée** avec :

- insertion rapide **où que ce soit** (O(1))
- suppression rapide (O(1))
- pas d'accès direct par index → **pas** list[i]

Exemple :

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

int main() {
   list<int> l = {1, 2, 3};
   l.push_front(0);

   for (int x : l)
      cout << x << " ";
}
```

Fonctions importantes :

| Fonction | Description |
|---|---|
| push_front() | ajoute en tête |
| push_back() | ajoute en fin |
| insert(it, x) | ajout avant l'itérateur |
| erase(it) | supprime un élément |
| sort() | tri intégré |

Complexité

| Opération | Complexité |
|---|---|
| Insertion / suppression | O(1) |
| Parcours | O(n) |
| Accès aléatoire | ✕ impossible |

**4. Le conteneur map (dictionnaire ordonné)**

std::map implémente une table associative ordonnée basée sur un arbre rouge-noir.

Caractéristiques :

- clé unique
- tri automatique (ordre croissant par défaut)
- accès en O(log n)

Exemple :

```
#include <map>
using namespace std;

int main() {
   map<string, int> age;
   age["Alice"] = 30;
   age["Bob"] = 25;

   for (auto& p : age)
      cout << p.first << " : " << p.second << endl;
}
```

Fonctions utiles :

| Fonction | Description |
|---|---|
| m[key] | accède ou crée un élément |
| insert({k,v}) | insère sans écraser |
| find(key) | retourne un itérateur |
| erase(key) | supprime |

**5. unordered_map (pour comparaison)**

À connaître car plus rapide dans de nombreux cas.

- basé sur un **hash**
- accès en O(1) en moyenne
- pas d'ordre des clés

unordered_map<string,int> um;

**6. Le conteneur stack (pile)**

std::stack applique le principe **LIFO** (Last In – First Out).

Il est basé par défaut sur deque.

Exemple :

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

int main() {
   stack<int> p;
   p.push(10);
   p.push(20);

   cout << p.top();  // 20
   p.pop();
}
```

Fonctions :

| Méthode | Fonction |
|---|---|
| push(x) | empiler |
| pop() | dépiler |
| top() | lire le sommet |
| empty() | vide ? |
| size() | taille |

**7. Les itérateurs dans la STL**

Les itérateurs standardisent le parcours des conteneurs :

```
vector<int> v = {1,2,3};

for (auto it = v.begin(); it != v.end(); ++it)
    cout << *it;
```

Types d'itérateurs courants :

| Type | Exemples |
|---|---|
| Random access | vector, deque |
| Bidirectionnel | list, map |
| Forward | forward_list |

**8. Les algorithmes standards <algorithm>**

Les algorithmes STL sont **indépendants des conteneurs**, ils fonctionnent avec les itérateurs.

**8.1. Tri : sort**

```
sort(v.begin(), v.end());
```

Tri décroissant :

```
sort(v.begin(), v.end(), greater<int>());
```

**8.2. Recherche : find**

```
auto it = find(v.begin(), v.end(), 3);
if (it != v.end()) cout << "trouvé";
```

**8.3. Comptage : count**

```
int c = count(v.begin(), v.end(), 10);
```

**8.4. Transformation : transform**

```
transform(v.begin(), v.end(), v.begin(),
        [](int x){ return x*x; });
```

8.5. Itérations : for_each

```
for_each(v.begin(), v.end(),
        [](int x){ cout << x << " "; });
```