



جامعة عبد المالك السعدي
Université Abdelmalek Essadi



Programmation Orientée Objet C++

2023-2024

1ère Année Génie Informatique

Prof : Anouar RAGRAGUI

Plan du cours

- ❑ Chapitre 1: Introduction à l'orienté objet
- ❑ Chapitre 2: Spécificité de C++
- ❑ Chapitre 3: Classes et objets
- ❑ Chapitre 4: Fonctions et classes amies && Surdéfinition des opérateurs
- ❑ Chapitre 5: Héritage && Polymorphisme
- ❑ Chapitre 6: Les Templates
- ❑ **Chapitre 7: Notion de la bibliothèque STL**
- ❑ Chapitre 8: Gestion des Exceptions
- ❑ Chapitre 9: Gestion des bases de données
- ❑ Chapitre 10: Introduction aux interfaces graphiques avec C++ (QT)

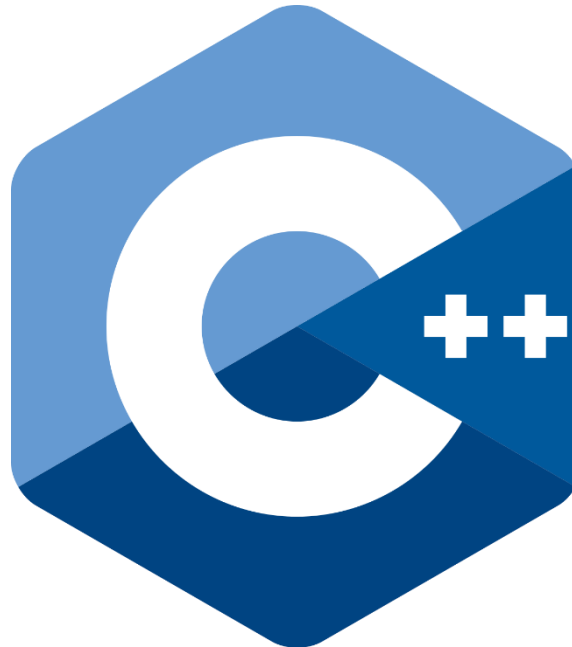


جامعة عبد المالك السعدي
Université Abdelmalek Essaadi



Programmation Orientée Objet C++

Chapitre 7 : Notion de la bibliothèque STL



Plan

- Programmation générique
- Standard Template Library

- La **programmation générique** est une technique qui permet de créer des algorithmes et des structures de données qui peuvent être utilisés **avec différents types de données** sans avoir à réécrire le code pour chaque type de données.
- Les tâches étudiés par les programmes sont souvent les mêmes.
- **Exemples :**
 - ▣ Gérer une liste d'étudiants, une liste de livres, une liste de voitures, ...
 - ▣ Compter dans une liste les éléments
 - ▣ Effacer le 'i'ème élément d'une liste :
 - ▣ etc.
- **Seul la nature des données change !**
- Afin **d'éviter de reprogrammer tout à chaque fois**, la librairie standard C++ fournit des outils facilitant l'intégration des données et des algorithmes à y appliquer "**Programmation générique**".
- Cette partie de la bibliothèque standard s'appelle la "**S.T.L**" : **Standard Template Library**

□ La bibliothèque standard du C++

▣ Formée de trois grandes composantes:

- STL — Standard Template Library
- Classes usuelles — String, flux d'entrée/sortie (cin, cout, cerr, etc.)
- Bibliothèque du langage C.

▣ Il ne faut confondre Bibliothèque standard du C++ et STL

□ Qu'est ce que STL?

▣ Une bibliothèque de classes et d'algorithmes contenant:

- des patrons de classes qui implémentent les structures de données usuelles,
- des patrons d'algorithmes qui implémentent les algorithmes classiques optimisés

□ Avantages de STL

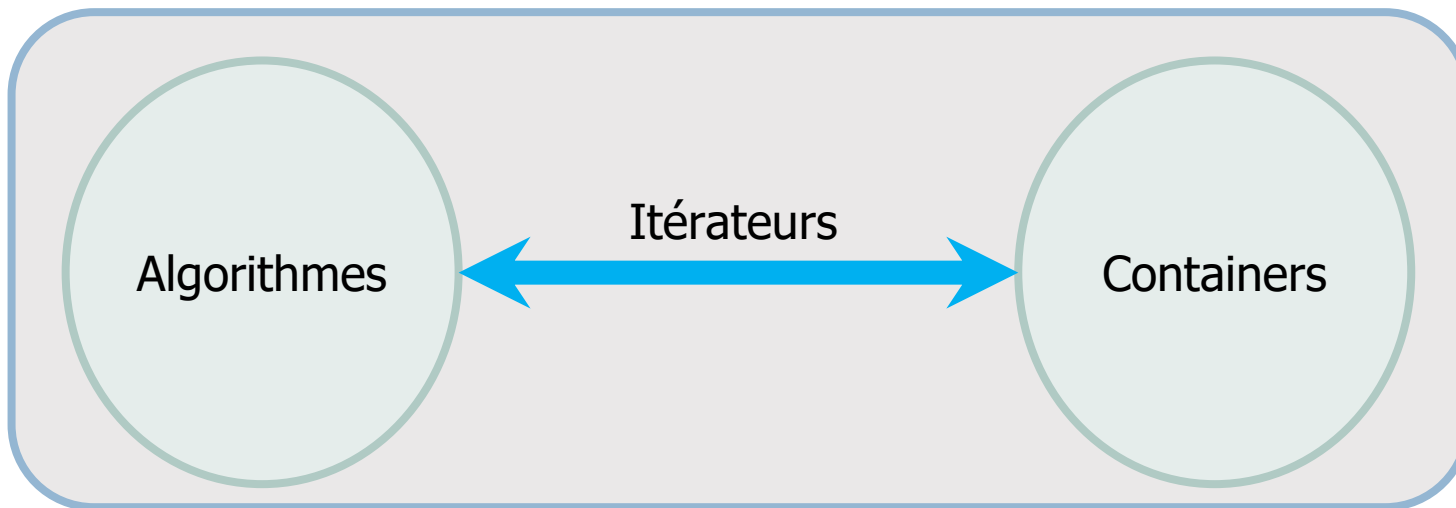
- Permet au programmeur de se focaliser sur les fonctionnalités de son application et d'éviter de reprogrammer tout à chaque fois.
- Fournit une meilleure efficacité : les structures de données et les algorithmes usuelles sont optimisés et validés.
- Favorise la réutilisabilité du code.

□ STL comporte les éléments suivants:

- ▣ **Conteneurs (container)** : Implémentation des structures de données les plus courantes avec leur allocation/désallocation mémoire automatique
- ▣ **Itérateurs (iterator)** : Permettent l'accès, le parcours et la manipulation des éléments des conteneurs
- ▣ **Algorithmes (algorithm)** : Implémentation d'algorithmes usuels tels que la recherche, le tri, etc.

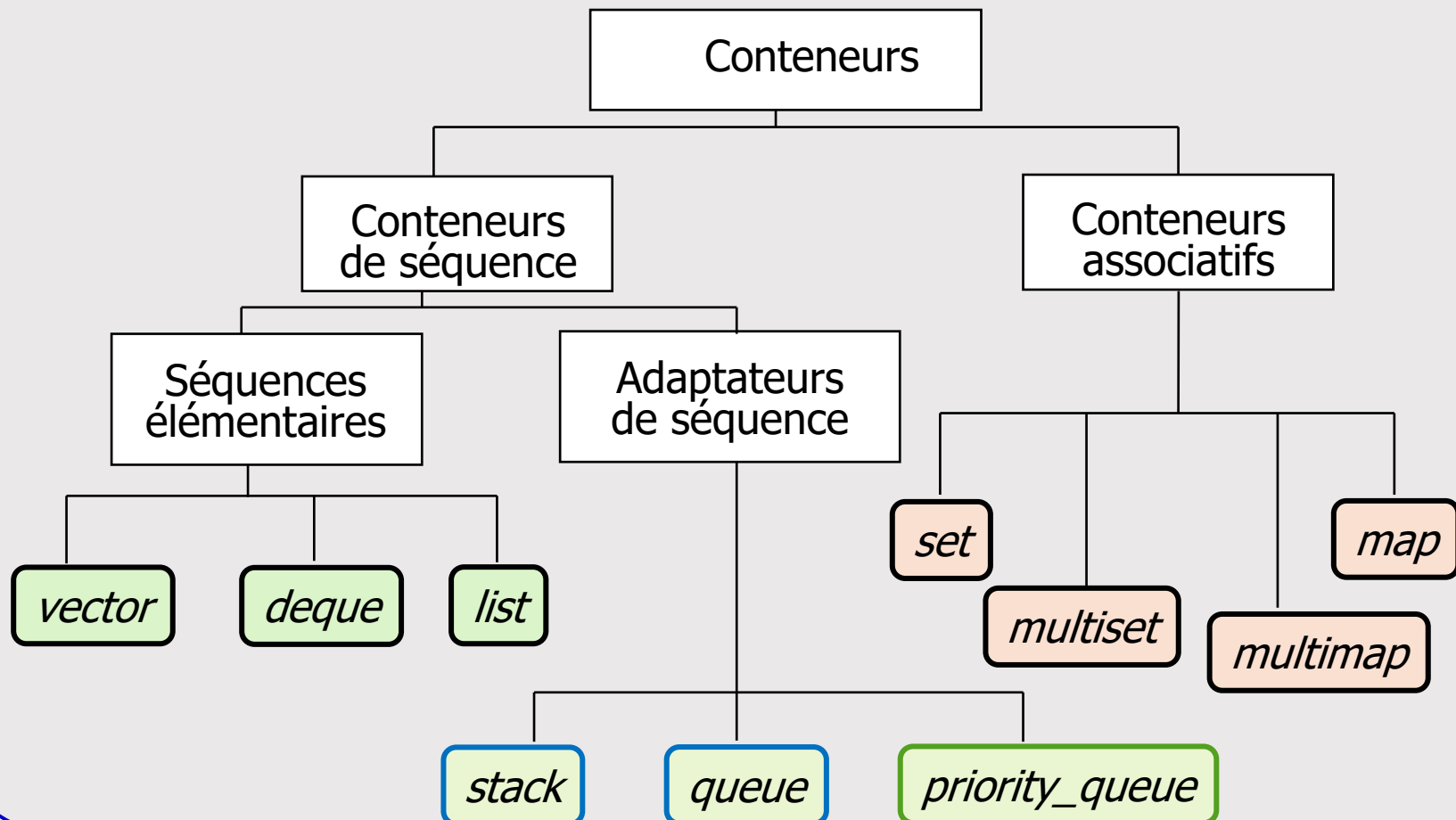
□ Séparation entre les éléments :

- ▣ Les **conteneurs** stockent les données sans savoir les algorithmes qui les manipulent
- ▣ Les **algorithmes** manipulent les données sans savoir le conteneur qui le contient.
- ▣ Les **itérateurs** assurent l'interaction entre les conteneurs et les algorithmes



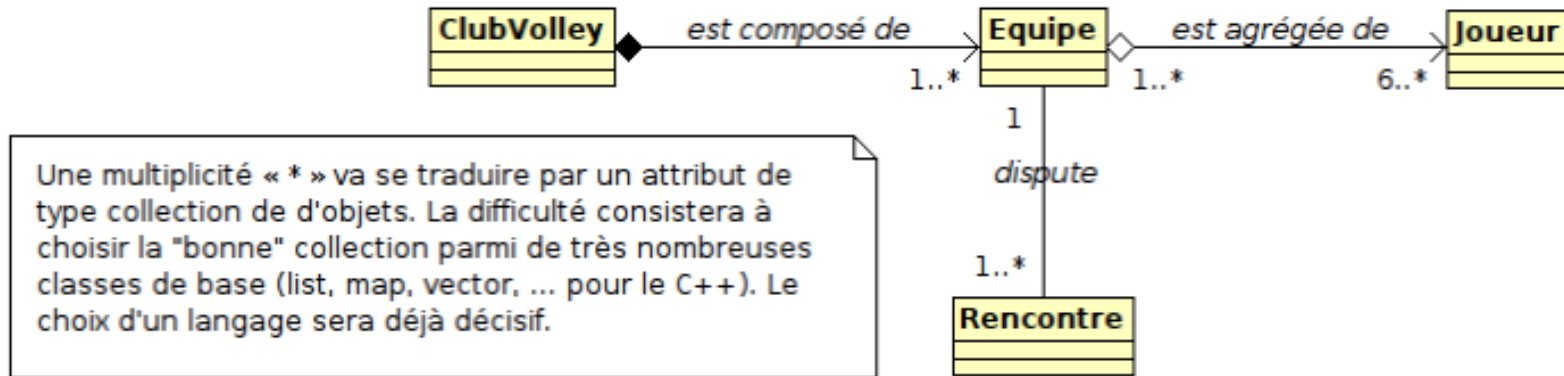
- La **STL** fournit un certain nombre de conteneurs pour gérer des collections d'objets :
 - ▣ les **tableaux** (**vector**), les **listes** (**list**), les **ensembles** (**set**), les **pires** (**stack**), et beaucoup d'autres ...
- Un **conteneur** (container) est un objet qui contient d'autres objets.
 - ▣ Il fournit un moyen de gérer les objets contenus
 - ▣ Assurer les opérations tel que : minimum, ajout, suppression, parfois insertion, tri, recherche, ...
- L'accès à ces objets qui dans le cas de la **STL** consiste très souvent en un **itérateur**.

□ STL comporte les éléments suivants:



- Les **conteneurs associatifs** sont des conteneurs qui stockent
 - ▣ les éléments dans un **ordre spécifique** en utilisant une **clé d'identification unique**.
 - ▣ Les éléments sont organisés par la clé de tri et l'accès aux éléments se fait en utilisant cette clé.
 - ▣ Les conteneurs associatifs les plus couramment utilisés sont le **map** et le **set**.
- Les **conteneurs de séquence** sont des conteneurs qui stockent
 - ▣ les éléments dans **un ordre linéaire** en utilisant une structure de données telle qu'un **tableau dynamique** ou une **liste chaînée**.
 - ▣ Les éléments sont stockés dans l'ordre où ils ont été ajoutés au conteneur et l'accès aux éléments se fait par leur position dans la séquence.
 - ▣ Les conteneurs de séquence les plus couramment utilisés sont le **vector**, la **liste** et la **deque**.
- Le choix du conteneur dépendra **des besoins spécifiques du projet** et de la manière dont les éléments doivent être stockés et accessibles.
 - ▣ Certains **conteneurs** ont des algorithmes plus ou moins efficaces pour accéder à un élément, pour l'insérer,...
 - ▣ Pour quantifier l'efficacité, on s'intéresse à son contraire, la **complexité**.

- **Exemple:** une liste d'articles à commander, une flotte de bateaux, les équipes d'un club, ...



- Bien qu'il soit possible de créer des tableaux d'objets, ce ne sera pas forcément la bonne solution. On préfère recourir à des collections parmi lesquelles les plus utilisées sont :
 - ▣ Java : ArrayList, HashMap, ...
 - ▣ C# : ArrayList, SortedList, HashTable ...
 - ▣ C++ (STL) : vector, list, map, set ...
 - ▣ Qt/C++ : QVector, QList, QMap ...

□ Le tableau dynamique

- Un **vector** est un **conteneur séquentiel** qui encapsule les **tableaux de taille dynamique**.

- ▣ Les éléments sont stockés de façon contigüe, ce qui signifie que les éléments sont accessibles non seulement via les itérateurs, mais aussi à partir des pointeurs classiques sur un élément.

Lien : <http://www.cplusplus.com/reference/vector/vector/>

- Il est particulièrement aisé d'**accéder directement** aux divers éléments par un **index**, et d'en ajouter ou en retirer à la fin.
- A la manière des tableaux de type C, l'espace mémoire alloué pour un objet de type **vector** est toujours continu, ce qui permet des algorithmes rapides d'accès aux divers éléments.

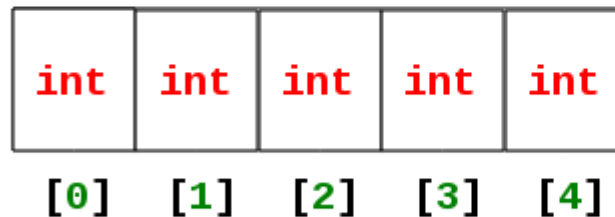
□ Le tableau dynamique

Le **type** des
éléments
composants le
vector

Le **nom** de la
variable vector

Le **nombre**
d'éléments
composants le
vector

`vector<int> vect(5);`



vector à 1
dimension
contenant 5
entiers (**int**)

Indice permettant
d'accéder à un élément du
vector

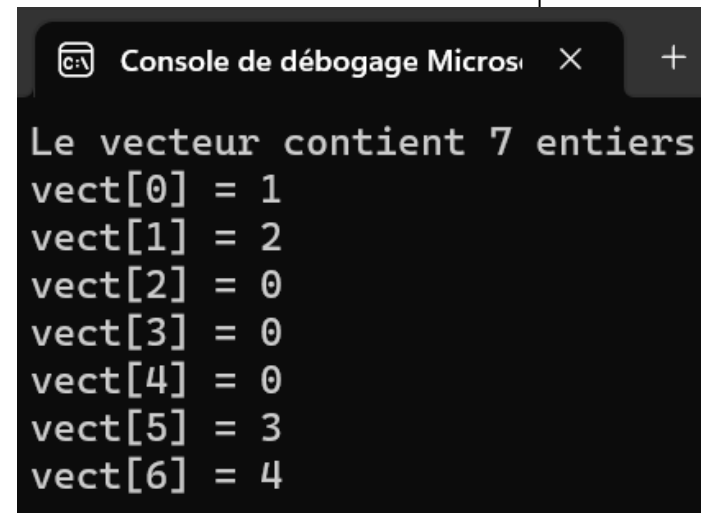
□ Exemple d'utilisation d'un **Vector**

```
#include<iostream>
#include<vector>
using namespace std;
void main() {
    vector<int> vect(5);

    vect[0] = 1;
    vect[1] = 2;

    vect.push_back(3);
    vect.push_back(4);
    vect.push_back(5);

    vect.pop_back();
    cout << "Le vecteur contient " << vect.size() << " entiers" << endl;
    for (int i = 0; i < vect.size(); i++)
        cout << "vect[" << i << "] = " << vect[i] << endl;
}
```



```
Le vecteur contient 7 entiers
vect[0] = 1
vect[1] = 2
vect[2] = 0
vect[3] = 0
vect[4] = 0
vect[5] = 3
vect[6] = 4
```

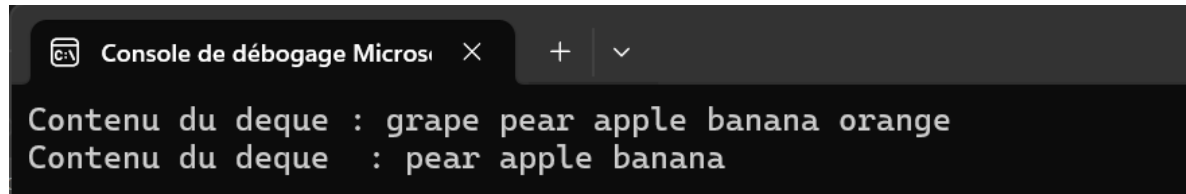

- Une « case » n'est accessible par l'opérateur [] que si elle a été allouée au préalable (sinon une erreur de segmentation est déclenchée).
- Il ne faut pas perdre de vue qu'une réallocation mémoire est coûteuse en terme de performances.
- Ainsi si la taille d'un vector est connue par avance, il faut autant que possible le créer directement à cette taille (voir méthodes resize et reserve).

- Il existe aussi un conteneur **deque** (*Double Ended QUEue*) qui s'utilise de la même façon que **vector** à deux différences :
 - ▣ deque est optimisé pour que les éléments soient ajoutés ou retirés à la fin (`push_back` et `pop_back`) ou **au début** (`push_front` et `pop_front`)
 - ▣ deque ne stockent pas (forcément) les éléments de façon contigüe, il est donc plus efficace en terme de réallocation mémoire importante
- La **STL** possède des adaptateurs de conteneurs (réduisant les possibilités d'un vector à quelques fonctionnalités) :
 - ▣ stack (pile), queue (file) et priority_queue (file à priorités).

□ Exemple:

```
#include <iostream>
#include <deque>
using namespace std;
int main() {
    deque<string> dq;
    dq.push_back("apple");
    dq.push_back("banana");
    dq.push_back("orange");
    dq.push_front("pear");
    dq.push_front("grape");

    cout << "Contenu du deque : ";
    for (const auto& elem : dq)
        cout << elem << " ";
    cout << endl;
    dq.pop_back();
    dq.pop_front();
    cout << "Contenu du deque : ";
    for (const auto& elem : dq)
        cout << elem << " ";
    cout << endl;
    return 0;
}
```



```
Console de débogage Micros...
Contenu du deque : grape pear apple banana orange
Contenu du deque : pear apple banana
```

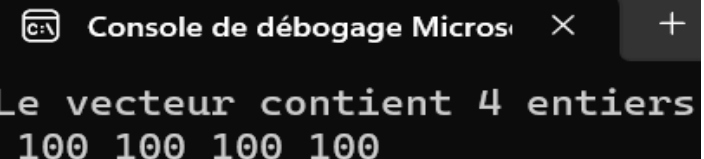
- Les **itérateurs** (*iterator*) sont une généralisation des **pointeurs** : ce sont des **objets qui pointent sur d'autres objets**.

Lien : <http://www.cplusplus.com/reference/iterator/>

- Comme son nom l'indique, les itérateurs sont utilisés pour **parcourir une série d'objets** de telle façon que si on incrémente l'itérateur, il désignera l'objet suivant de la série.

```
#include<iostream>
#include<vector>
using namespace std;
void main() {
    vector<int> vect(4,100);
    cout << "Le vecteur contient " << vect.size() << " entiers" << endl;

    for(vector<int>::iterator it=vect.begin();it!=vect.end();++it)
        cout << " " << *it ;
}
```



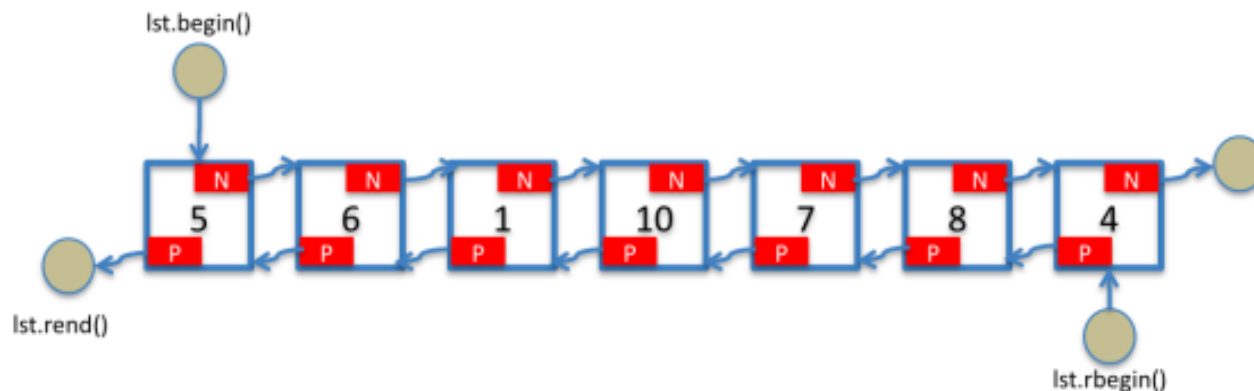
```
Le vecteur contient 4 entiers
100 100 100 100
```

- Il existe aussi un itérateur inversé : **reverse_iterator**

Lien : http://www.cplusplus.com/reference/iterator/reverse_iterator/

- La classe **list** fournit une structure générique de **listes doublement chaînées** (c'est-à-dire que l'on peut parcourir dans les deux sens) pouvant éventuellement contenir des doublons.

Lien : <http://www.cplusplus.com/reference/list/list/>



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

void main() {
    list<int> lst;

    lst.push_back(1);
    lst.push_back(2);
    lst.push_back(3);
    lst.push_back(4);
    lst.push_back(5);
    lst.push_back(6);
    lst.push_back(7);
    lst.push_back(8);
    lst.pop_back();

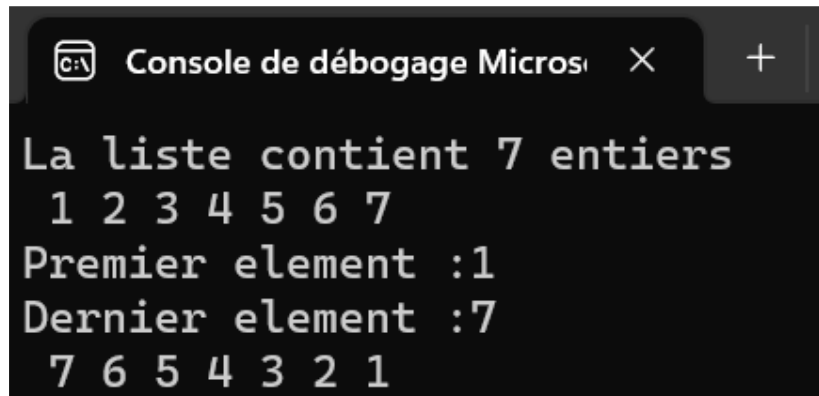
    cout << "La liste contient " <<
    lst.size() << " entiers" << endl;
```

```
for(list<int>::iterator
it=lst.begin();it!=lst.end();++it)
    cout << " " << *it ;
    cout << endl;

cout << "Premier element :" << lst.front() << endl;

cout << "Dernier element :" << lst.back() << endl;

for (list<int>::reverse_iterator rit = lst.rbegin();
rit != lst.rend(); ++rit)
    cout << " " << *rit;
}
```



```
La liste contient 7 entiers
1 2 3 4 5 6 7
Premier element :1
Dernier element :7
7 6 5 4 3 2 1
```

□ La table associative (map)

- Une **table associative** map permet d'**associer une clé à une donnée**.

Lien : <http://www.cplusplus.com/reference/map/map/>

- La **map** prend donc au moins deux paramètres :

- le **type de la clé** (dans l'exemple ci-dessous, une chaîne de caractères string)

values	
AL	Alabama
AK	Alaska
AZ	Arizona
AR	Arkansas
CA	California
CO	Colorado
...	...
keys	

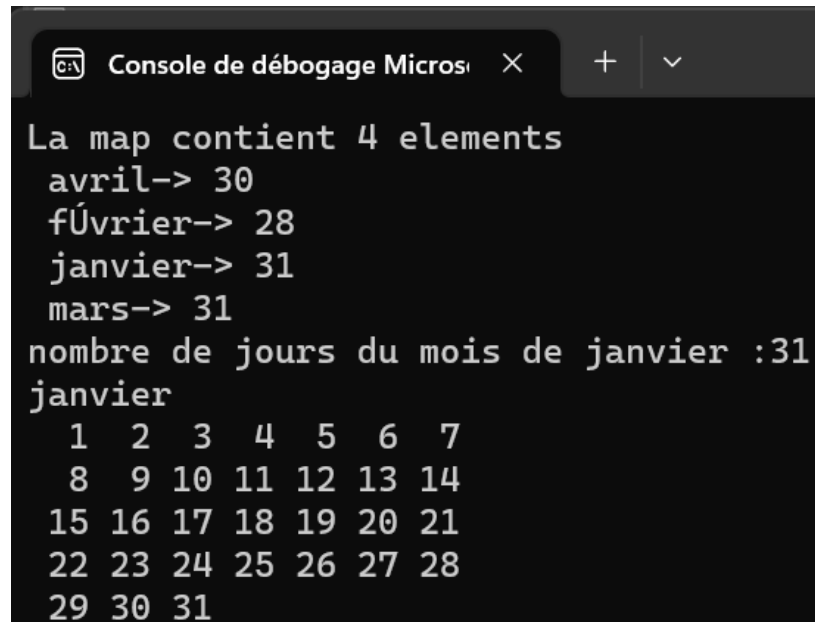
- Le tableau associatif (aussi appelé dictionnaire ou table d'association) peut être vu comme une généralisation du tableau
- Le tableau traditionnel associe des entiers consécutifs à des valeurs d'un certain type, le tableau associatif associe des valeurs d'un type arbitraire à des valeurs d'un autre type

```
#include<iostream>
#include<iomanip>
#include<string>
#include<map>
using namespace std;
void main() {
    map <string, unsigned int> nbJoursMois;
    nbJoursMois["janvier"] = 31;
    nbJoursMois["février"] = 28;
    nbJoursMois["mars"] = 31;
    nbJoursMois["avril"] = 30;

    cout << "La map contient " << nbJoursMois.size() << " elements" << endl;
    for (map<string, unsigned int>::iterator it = nbJoursMois.begin(); it != nbJoursMois.end();
        ++it)
        cout << " " << it->first << "-> " << it->second<<endl;

    cout << "nbr de jours du mois de janvier :" << nbJoursMois.find("janvier")->second << endl;
    cout << "janvier" << endl;
    for (int i = 1; i <= nbJoursMois["janvier"]; i++) {
        cout << setw(3)<<i;
        if (i % 7 == 0)
            cout << endl;
    }
}
```


□ Affichage



```
Console de débogage Micros... X + v
La map contient 4 elements
avril-> 30
février-> 28
janvier-> 31
mars-> 31
nombre de jours du mois de janvier :31
janvier
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

□ Un ensemble d'éléments (set)

- La **classe set** est un conteneur qui stocke des éléments uniques **suivants un ordre spécifique**
- Les ensembles **set** sont généralement mis en œuvre dans les arbres binaires de recherche.

Lien : <http://www.cplusplus.com/reference/set/set/>

```
#include<iostream>
#include<set>
using namespace std;
void main() {
int Entiers1[] = { 75 ,23 ,65 ,42 ,13 ,100 };
int Entiers2[] = { 75 ,23 ,75 ,23 ,13 };
set <int> ensemble1(Entiers1,Entiers1+6);
set <int> ensemble2(Entiers2, Entiers2 + 5);

cout << "l'ensemble set 1 contient";
for (set<int>::iterator it = ensemble1.begin(); it != ensemble1.end(); ++it)
    cout << " " << *it;

cout << endl;
cout << "l'ensemble set 2 contient";
for (set<int>::iterator it = ensemble2.begin(); it != ensemble2.end(); ++it)
    cout << " " << *it;
}
```



Console de débogage Micros



```
l'ensemble set 1 contient 13 23 42 65 75 100
l'ensemble set 2 contient 13 23 75
```

□ Une paire d'éléments (pair)

- Une **paire** est une **structure contenant deux éléments éventuellement de types différents**.

Lien : <http://www.cplusplus.com/reference/utility/pair/>

- Certains algorithmes de la STL (find par exemple) retournent des paires (position de l'élément trouvé et un booléen indiquant s'il a été trouvé).
- En pratique, il faut voir les classes conteneurs de la STL comme des « legos » (briques logicielles) pouvant être imbriqués les uns dans les autres. Ainsi, on pourra tout à fait manipuler un vector de pair, etc ...

```
#include<iostream>
#include<vector>
#include<map>
#include<set>
using namespace std;
void main() {
    pair <char, int> c1 = make_pair('B', 2);
    pair <char, int> c2 = make_pair('J', 1);

    cout << "un coup en " << c1.first << " , " << c1.second<<endl;

    pair <int, int> p;

    p.first = 1;
    p.second = 2;

    cout << "un coup en " << p.first << " , " << p.second << endl;

    vector<pair<char, int>> tableauCoups(2);

    tableauCoups[0] = c1;
    tableauCoups[1] = c2;
    cout << " le vector contient " << tableauCoups.size() << " coups " << endl;
```

```
for (vector<pair<char,int>>::iterator it = tableauCoups.begin(); it != tableauCoups.end(); ++it)
    cout << " " << it->first << " , " << it->second<<endl;

map<pair<char, int>, bool> mapCoups;
mapCoups[c1] = true;
mapCoups[c2] = false;

cout << " la map contient " << mapCoups.size() << " coups " << endl;

for (map<pair<char, int>,bool>::iterator it = mapCoups.begin(); it != mapCoups.end(); ++it)
    cout << " " << it->first.first << " , " << it->first.second <<" ->"<<it->second<< endl;

set<pair<char, int>> ensembleCoups;
ensembleCoups.insert(c1);
ensembleCoups.insert(c2);

cout << " l'ensemble set contient " << ensembleCoups.size() << " coups " << endl;

for (set<pair<char, int>>::iterator it = ensembleCoups.begin(); it != ensembleCoups.end(); ++it)
    cout << " " << it->first << " , " << it->second << endl;
}
```

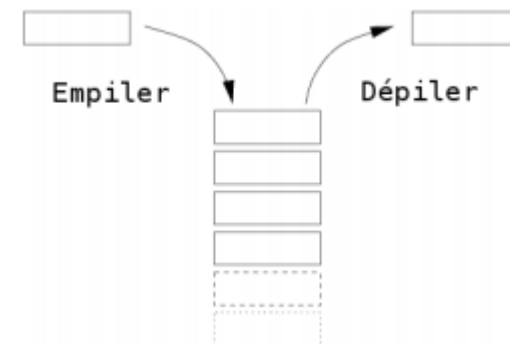
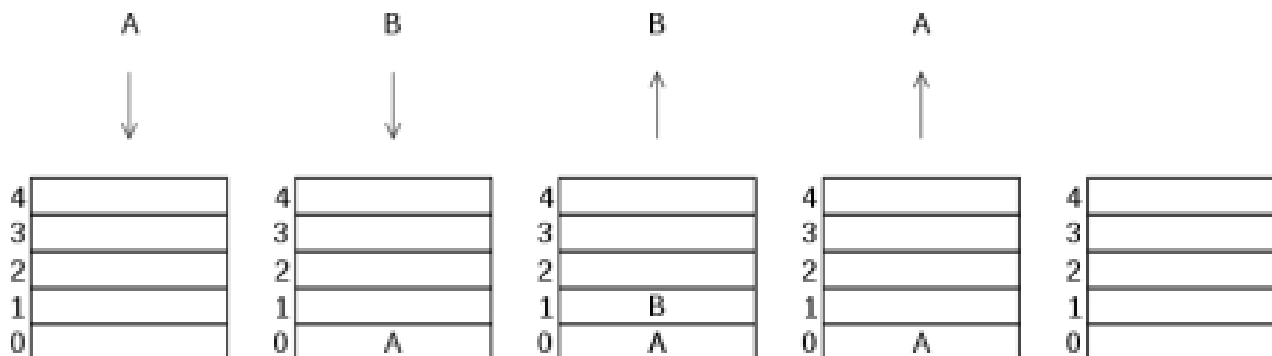
□ Affichage

```
Console de débogage Micros... X + v
un coup en B , 2
un coup en 1 , 2
le vector contient 2 coups
B , 2
J , 1
la map contient 2 coups
B , 2 ->1
J , 1 ->0
l'ensemble set contient 2 coups
B , 2
J , 1
```

□ La pile (stack)

- Une **pile** (« **stack** » en anglais) est une **structure de données basée sur le principe « Dernier arrivé, premier sorti », ou LIFO (Last In, First Out)**, ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.
- Le fonctionnement est celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.

Lien : <http://www.cplusplus.com/reference/stack/stack/>



□ La pile (stack)

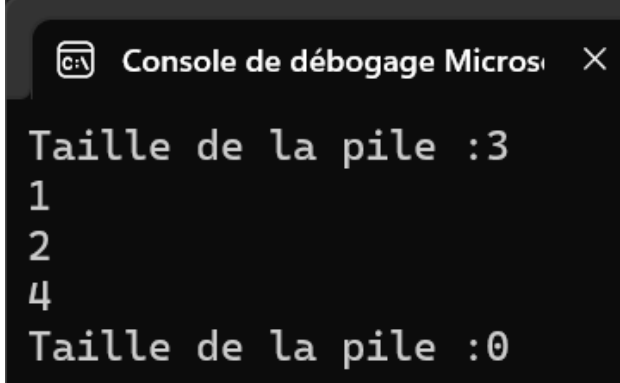
- Une **pile** est utilisée en général pour **gérer un historique de données** (pages webs visitées, ...) ou **d'actions** (les fonctions « Annuler » de beaucoup de logiciels par exemple)
- La pile est utilisée aussi pour tous les paramètres d'appels et les variables locales des fonctions dans les langages compilés.
- Voici quelques fonctions communément utilisées pour manipuler des **piles** :
 - « **Empiler** » : ajoute ou dépose un élément sur la pile
 - « **Dépiler** » : enlève un élément de la pile et le renvoie
 - « **La pile est-elle vide ?** » : renvoie « vrai » si la pile est vide, « faux » sinon
 - « **La pile est-elle pleine ?** » : renvoie « vrai » si la pile est pleine, « faux » sinon
 - « **Nombre d'éléments dans la pile** » : renvoie le nombre d'éléments présents dans la pile
 - « **Taille de la pile** » : renvoie le nombre maximum d'éléments pouvant être déposés dans la pile
 - « **Quel est l'élément de tête ?** » : renvoie l'élément de tête (le sommet) sans le dépiler

- ▶ Présentation
- ▶ Modèle de base
- ▶ Conteneurs
- ▶ Vector
- ▶ Iterator
- ▶ List
- ▶ Map
- ▶ Set
- ▶ Pair
- ▶ **Stack (Exemple)**
- ▶ Queue
- ▶ Remarques

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

void main() {
    stack<int>pile;
    pile.push(4);
    pile.push(2);
    pile.push(1);

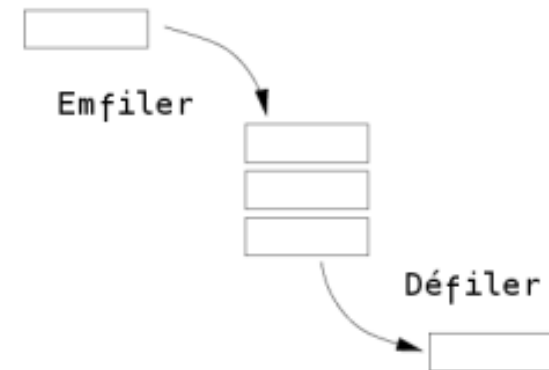
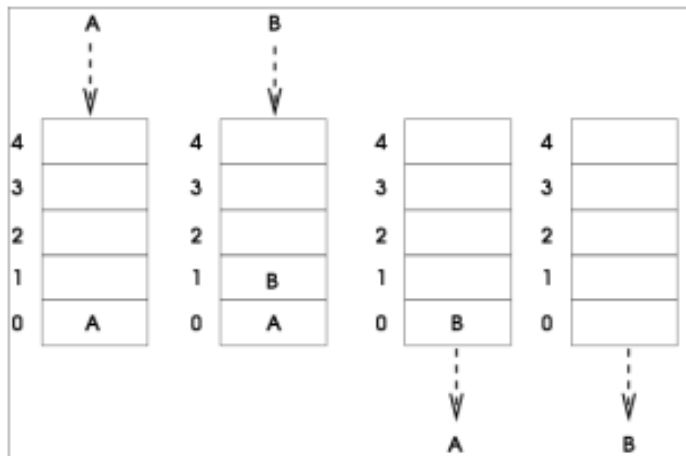
    cout << "Taille de la pile :" << pile.size() << endl;
    while (!pile.empty()) {
        cout << pile.top() << endl;
        pile.pop();
    }
    cout << "Taille de la pile :" << pile.size() << endl;
}
```



```
Console de débogage Micros X
Taille de la pile :3
1
2
4
Taille de la pile :0
```

❑ La file (queue)

- ❑ Une **file** (ou **file d'attente**, « **queue** » en anglais) est une **structure de données basée sur le principe « Premier arrivé, premier sorti », ou FIFO (First In, First Out)**, ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés.



- ❑ Le fonctionnement est celui d'une salle d'attente : les premières personnes arrivées sont les premières personnes à sortir de la file.

□ La file (queue)

- Une liste chaînée dont on n'utilise que les opérations ajouterQueue et retirerTête constitue une queue. Si la queue se base sur un tableau, la structure enregistre deux indices, l'un correspondant au dernier arrivé, l'autre au prochain à sortir.
- Une **file** est utilisée en général pour mémoriser temporairement des transactions qui doivent attendre pour être traitées (notions de serveur et de *spool*) et pour créer toutes sortes de mémoires tampons (« *buffers* »).
- Voici quelques fonctions communément utilisées pour manipuler des **files** :
 - ▣ « **Emfiler** » : ajoute un élément dans la file (*enqueue*)
 - ▣ « **Défiler** » : renvoie le prochain élément de la file, et le retire de la file (*dequeue*)
 - ▣ « **La file est-elle vide ?** » : renvoie « vrai » si la file est vide, « faux » sinon
 - ▣ « **Nombre d'éléments dans la file** » : renvoie le nombre d'éléments présents dans la file

```
#include <iostream>
#include<queue>
using namespace std;
int main() {
    queue<int> maQueue; // déclare une queue d'entiers vide

    maQueue.push(10); // ajoute 10 à la fin de la queue
    maQueue.push(20); // ajoute 20 à la fin de la queue
    maQueue.push(30); // ajoute 30 à la fin de la queue

    // Parcours des éléments de la queue avec une boucle
    while (!maQueue.empty()) {
        cout << maQueue.front() << " "; //affiche le premier élément de la queue
        maQueue.pop(); // retire le premier élément de la queue
    }
    cout << endl;

    return 0;
}
```



Console de débogage Micros



10 20 30

❑ La file (queue)

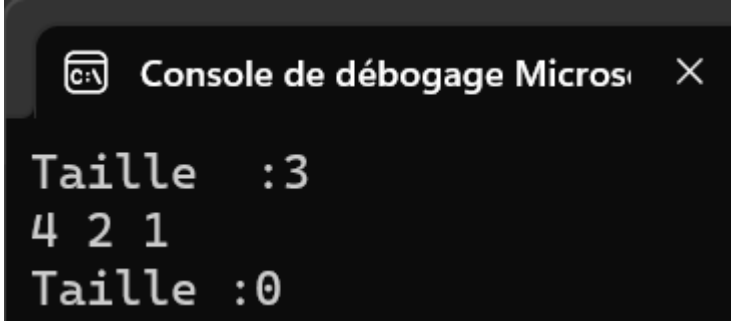
- ❑ Il existe aussi des files à priorité qui permettent de récupérer l'élément de plus grande valeur.
- ❑ Elles permettent d'implémenter efficacement des planificateurs de tâches, où un accès rapide aux tâches d'importance maximale est souhaité.
- ❑ Les files à priorité sont utilisés dans certains algorithmes de tri et de recherche.

```
#include <iostream>
#include<queue>
using namespace std;
int main() {
    priority_queue<int> file;

    file.push(1);
    file.push(4);
    file.push(2);

    cout << "Taille :" << file.size()<<endl;
    while (!file.empty()) {
        cout << file.top() << " ";
        file.pop();
    }
    cout << endl;
    cout << "Taille :" << file.size() << endl;

    return 0;
}
```



```
C:\N Console de débogage Micros X
Taille :3
4 2 1
Taille :0
```

□ Notion d 'algorithmes:

- A tous ces conteneurs sont associés des patrons de fonctions qui effectuent des opérations.
- Par exemple la fonction sort (algorithme de tri)

```
int t[] = { 5,2,1,3,4 };  
list <int> l(t, t + 5);  
l.sort();
```

□ résultat:1,2,3,4,5

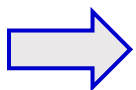
□ Certains algorithmes sont applicables, d 'autres non

- par exemple, l 'algorithme de tri est applicable si une relation d 'ordre a été définie sur les éléments du conteneur **list**.

□ Générateurs d'opérateurs.

▣ La bibliothèque peut générer:

- l'opérateur `!=` à partir de l'opérateur `==`
- les opérateurs `>`, `>=`, `<=` à partir de l'opérateur `<`;



Par conséquent, dans une classe, il suffit de la munir des deux seuls opérateurs `==` et `<`.

□ Constructeurs et affectation.

- ▣ Tous les conteneurs possèdent un constructeur par copie et un opérateur d'affectation.

```
int t[5] = { 1,2,3,4,5 };  
list<int> l(t, t + 5);  
vector<int> v(l.begin(), l.end());  
vector<int> v(l.rbegin(), l.rend()); //itérateurs inverses
```

□ Fonctionnalités communes aux conteneurs séquentiels (**vector**, **deque**, **list**)

- **assign()** modification du contenu du conteneur

```
vector<int> v{2,1,4,5};  
list<int> l;  
l.assign(v.begin(), v.end());
```

- **clear()** vide le conteneur
- **swap()** échange le contenu de 2 conteneurs de même type.
- **insert(position,valeur)**, **insert(position,NbFois,valeur)**, **insert(debut,fin,valeur)**.
- **erase(position)**, **erase(debut, fin)** pour supprimer un élément ou un intervalle d'éléments.
- **push_back()** et **pop_back()** insertion et suppression sur le dernier élément.
- **push_front()** et **pop_front()** uniquement pour les conteneurs **deque** et **list**. (ajout et suppression sur le premier élément)

□ Fonctionnalités propres au conteneur **list**

- **remove**(valeur) supprime les éléments égaux à valeur.
- **remove_if**(predicat)
- **sort**() et **sort**(predicat)
- **unique**() : supprime les doublons.
- **merge**(liste) et **merge**(liste, predicat): injecte les éléments de la liste « liste » dans la liste courante (en conservant l'ordre définit par le prédicat)
- **splice**(pos, liste) : injecte le contenu de la liste « liste » dans la liste courante à partir de la position « pos »