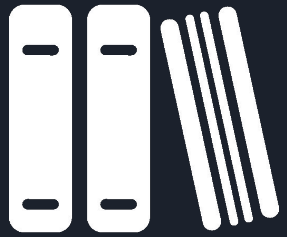
# Les Conteneurs STD

# Les conteneurs

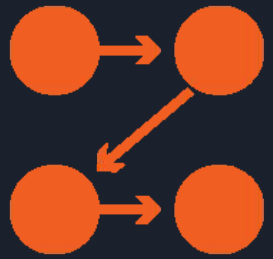

Différentes **classes C++** servant à **stocker** des collections de données.



Elles font parties de la bibliothèque standard **STD**.

# Les conteneurs

Deux types de conteneurs :



**Les séquences**
Données associées à un **index**.

**Les conteneurs associatifs**
Données associées à une **clé de recherche**.

# Les séquences

**vector** : Tableau à taille variable (ajouts en fin de tableau)

**list** : Liste **chaînée** de données

**stack** : Pile de données **LIFO**

**queue** : Queue de données **FIFO**

**deque** : Tableau à **taille variable** (ajouts en début ou fin)

**priority_queue** : Queue de données avec **priorité**

# Les conteneurs associatifs

**map** : Associations de données **clé-valeur**

**set** : Données triées **automatiquement**

**multimap** : map autorisant les **clés doublons**

**multiset** : set autorisant les **clés doublons**

# Type de données

**Les conteneurs** de la **STD** sont des classes dites **templates**.

Un **type de données** est choisi lors de la **déclaration** d'une variable conteneur.

**Toutes les données stockées dans ce conteneur seront de ce type.**

```
std::list<float> myList; // liste de float
```

# Vector

Le **tableau à taille dynamique**, ou **std::vector**, est une structure stockant des éléments **ordonnés contigus** en mémoire (tableau dynamique).

```
std::vector<char> myVector(5, 'a');
```

ou

```
std::vector<char> myVector { 'a','b','c' };
```

# Vector : Ajout / Suppression

Il est possible **d'ajouter** ou de **supprimer** des éléments en fin de tableau.

```
std::vector<char> myVector(3, 'a');
myVector.pop_back();
myVector.push_back('b');
myVector.resize(6,'c');
// {'a','a','b','c','c','c'}
```

# Vector : Réallocation

`std::vector` réserve une **zone mémoire** de taille **supérieure** à son besoin.

Si cette taille est dépassée lors d'un changement de taille, alors le tableau est automatiquement réalloué et les variables copiées.

```
std::vector<int> myVector(5);
myVector.resize(20);
```

# Vector : Accès

Pour accéder à un **élément du tableau**, on utilise simplement **les crochets** `[ ]` :

```
std::vector<int> myVector {5, 8, 3};
int i = myVector[1]; // i = 8
```

**Attention : Les adresses mémoires des données stockées sont susceptibles de changer lors des réallocations !**

# Vector : Performances

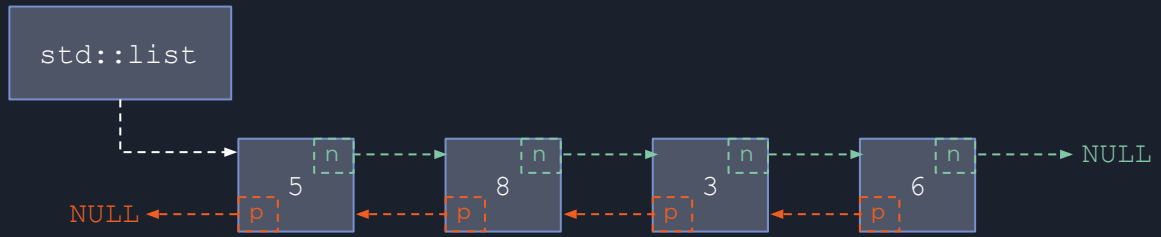
Performant en **accès par index**
(éléments contigus en mémoire)

Peu performant lors **des gros changements de tailles**
(réallocation mémoire)

# Liste

**La liste** (doublement chaînée), ou `std::list`, est une structure stockant des éléments **ordonnés non-contigus** en mémoire.

```
std::list<int> myList;
```



# Liste : Ajout/Suppression

Pour **ajouter ou de supprimer** des éléments en début et fin de liste, on utilise :

```
std::list<int> myList;
myList.push_back(5); // {5}
myList.push_back(8); // {5,8}
myList.push_front(3); // {3,5,8}

myList.pop_front(); // {5,8}
myList.pop_back(); // {5}
```

# Itérateur

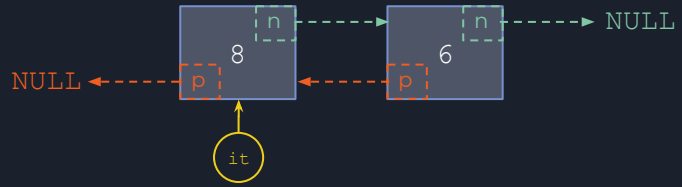
Pour **parcourir une liste**, on utilise un **itérateur** :

```
std::list<int> myList;
myList.push_back(1);
myList.push_back(3);

std::list<int>::iterator it = myList.begin();
while (it != myList.end()) {
  printf("%d ", (*it));
  it++;
}
```

# Itérateur de début

La fonction `begin()` renvoie un **itérateur** positionné sur le premier élément du conteneur.



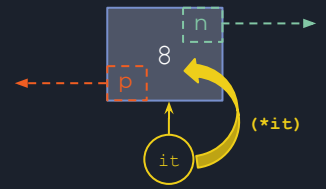

```
std::list<int> myList;
myList.push_back(8);
myList.push_back(6); // {8,6}

// Itérateur placé sur le 1er élément
std::list<int>::iterator it = myList.begin();
printf("%d ", (*it)); // Affiche : 8
```

# Déréférencement d'itérateur

**L'opérateur de déréférencement *** permet **d'accéder** à l'élément sur lequel **l'itérateur est positionné**.

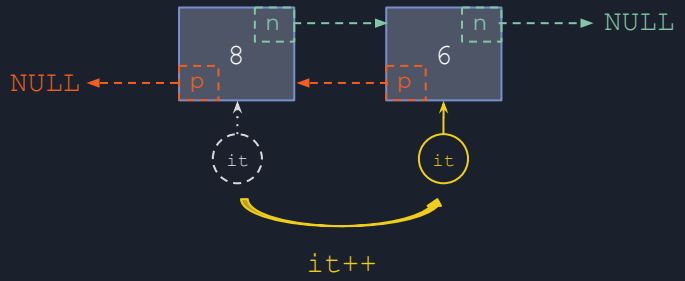
```
std::list<int>::iterator it = myList.begin();
printf("%d ", (*it));
// Affiche la valeur de l'élément sur lequel
'it' est positionné.
```



# Déplacement d'itérateur

On déplace un itérateur avec les opérateurs :

- `++` : passe l'itérateur sur l'élément suivant.
- `--` : passe l'itérateur sur l'élément précédent.
- `std::advance(it, n)` : déplace l'itérateur de n éléments.



# Liste : ajout par itérateur

**La fonction** `insert()` permet **d'ajouter** un élément **avant l'élément** indiqué par un itérateur.

```
std::list<int> myList;
myList.push_back(1);
myList.push_back(3);
myList.push_back(5); // {1,3,5}

std::list<int>::iterator it = myList.begin();
it++; // Itérateur placé sur le 2ème élément
myList.insert(it,8); // {1,8,3,5}
```

# Liste : suppression par itérateur

**La fonction** `erase()` permet de **supprimer l'élément** indiqué par un itérateur.

```
std::list<int> myList;
myList.push_back(1);
myList.push_back(3);
myList.push_back(5); // {1,3,5}

std::list<int>::iterator it = myList.begin();
it++; // Itérateur placé sur le 2ème élément
myList.erase(it); // {1,5}
```

# Liste : itérateurs invalidés

Lorsqu'une liste **change de taille**, tous les itérateurs sur cette liste **sont dits invalidés**.

**Un itérateur invalidé ne doit plus être utilisé.**

```
std::list<int>::iterator it = myList.begin();
it++; // Itérateur placé sur le 2ème élément

myList.erase(it);
// myList a changé de taille. Ne plus utiliser it !
```

# Liste : itérateurs invalidés

Pour éviter d'utiliser **des itérateurs invalidés**, les fonctions `insert()` et `erase()` retournent de **nouveaux itérateurs valides.**

```
std::list<int>::iterator it = myList.begin();
it++; // Itérateur placé sur le 2ème élément

it = myList.erase(it);
// 'it' est valide et est positionné sur le nouveau deuxième élément.
```

# Liste : clear

**La fonction** `clear()` permet de **vider intégralement** un conteneur :

```
std::list<int> myList;
myList.push_back(1);
myList.push_back(3);
myList.push_back(5); // {1,3,5}

myList.clear(); // {}
```

# Liste : size

**La fonction** `size()` donne **le nombre d'éléments** dans un conteneur :

```
std::list<int> myList;
myList.push_back(1);
myList.push_back(3);
myList.push_back(5); // {1,3,5}

printf("%d\n", myList.size()); // Affiche : 3
```

# Liste : Performances

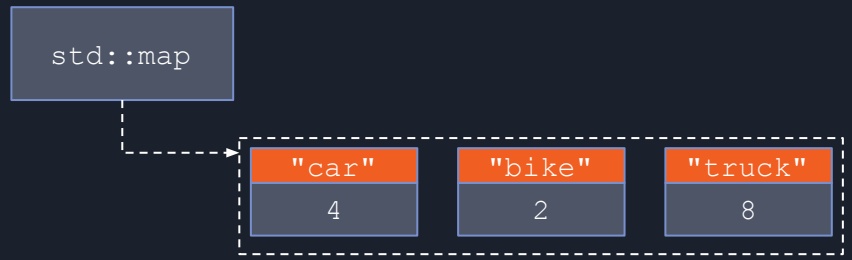
Performant en **parcours par itération** et pour **ajouter/supprimer** des éléments **en milieu de liste**.

Peu performant pour **accéder à un élément donné** (on doit parcourir **l'ensemble de la liste** pour accéder au **dernier élément**).

# Dictionnaire

**Le dictionnaire**, ou `std::map`, est une **structure** stockant des éléments associés à des **clés de recherche**.

```
std::map<std::string, int> myMap;
```



# Dictionnaire : Accès

L'opérateur `[ ]` permet **d'accéder à un élément** dans un dictionnaire en **recherchant sa clé**.

```
std::map<int, std::string> myMap;
myMap[5] = "Five";
myMap[8] = "Eight";
printf("%s", myMap[8]); // Affiche : Eight
```

**Attention : Si cette clé n'existe pas, une nouvelle association "clé-valeur" sera créée !**

# Dictionnaire : Itérateur

Il est possible de **parcourir un dictionnaire** avec un **itérateur** :

```
std::map<int, std::string> myMap;
myMap[5] = "Five";
myMap[8] = "Eight";

std::map<int, std::string>::iterator it = myMap.begin();
while (it != myMap.end()) {
  std::cout << "Key:" << (*it).first << std::endl;
  std::cout << "Value:" << (*it).second << std::endl;
  it++;
}
```

# Dictionnaire : Recherche

**La fonction** `find()` permet de **chercher une clé** dans un dictionnaire **sans créer de nouvelle association** dans le cas où la **clé n'existe pas** :

```
std::map<int, std::string> myMap;
myMap[5] = "Five";
myMap[8] = "Eight";

std::map<int, std::string>::iterator found = myMap.find(5);
if (found != myMap.end()) {
  std::cout << "Key:" << (*found).first << std::endl;
  std::cout << "Value:" << (*found).second << std::endl;
}
```

# Dictionnaire : Suppression

**La fonction** `erase()` permet de **supprimer** une association "clé-valeur" à partir **d'une clé** ou **d'un itérateur** :

```
std::map<std::string, int> myMap;
myMap["Everest"] = 8848;
myMap["K2"] = 8611;
myMap["Kangchenjunga"] = 8586;

myMap.erase("K2");
```

# Dictionnaire : Performances

En interne, **les clés d'un dictionnaire** sont converties et triées automatiquement dans une **table de hachage**.

**Cette conversion automatique** permet une **recherche très rapide** des clés, **quel que soit le nombre d'éléments** dans le dictionnaire.

# Dictionnaire : Performances

- Performant **en accès** par **recherche de clé**.
- Performant en **ajout/suppression d'associations**.
- Performant en **parcours itératif**.

En revanche, `std::map` **ne garantit pas de position** des éléments dans le conteneur.

# Range-based loop : List

La syntaxe **`for( : )`** **parcourt un conteneur** de manière simplifiée.

```
std::list<int> myList = { 0, 1, 2 };

for (int &ri : myList) { // ri est une référence.
     ri++; // Les valeurs de la liste sont modifiées.
}


for (int i : myList) { // i est une variable “copiée”.
     i++; // Les valeurs de la liste ne sont pas modifiées.
}
```

# Range-based loop : List

La syntaxe **`const &`** n'effectue pas de copies de valeurs et **interdit les modifications**.

```
for (int const &ri : myList) { // ri est une référence.
     ...
}
```

# Range-based loop : Map

Pour un **dictionnaire** :

```
std::map<int, std::string> myMap = { {0, "zero"}, {1, "un"} };


for (std::pair<const int, std::string> &keyValue : myMap) {
     keyValue.second += "!";
}


for (std::pair<const int, std::string> const &keyValue : myMap) {
     ...
}
```