

INTRODUCTION À LA BIBLIOTHÈQUE STL

DEFINITION DE LA STL

La STL (Standard Template Library) est une bibliothèque de classes et de fonctions. Elle fait partie de la bibliothèque standard du langage C++. La STL fournit des structures de données génériques et des algorithmes. Elle permet de manipuler des collections de données de manière efficace. Les principaux composants de la STL sont les conteneurs, les algorithmes et les itérateurs.

IMPORTANCE DE LA STL

La STL simplifie le développement en fournissant des solutions prêtes à l'emploi. Elle améliore la réutilisabilité et la maintenabilité du code. La STL est optimisée pour la performance. Elle permet de se concentrer sur la logique métier plutôt que sur les détails d'implémentation. Elle est largement utilisée et bien documentée.

AVANTAGES DE LA STL

- Réduction du temps de développement
- Code plus propre et plus lisible
- Performances optimisées
- Réutilisabilité accrue
- Large communauté et support

STRUCTURE DE LA STL

La STL est composée de trois principaux composants :

1. Conteneurs : Structures de données génériques.
2. Algorithmes : Fonctions pour manipuler les données.
3. Itérateurs : Objets pour parcourir les conteneurs.

CONTENEURS DE LA STL

Les conteneurs sont des structures de données génériques. Ils incluent des classes telles que `vector`, `map`, `stack`, et `queue`. Chaque conteneur a des caractéristiques et des usages spécifiques. Les conteneurs sont optimisés pour différentes opérations (insertion, suppression, accès).

ALGORITHMES DE LA STL

Les algorithmes sont des fonctions génériques pour manipuler les données. Ils incluent des opérations comme le tri, la recherche, et la transformation. Les algorithmes fonctionnent avec n'importe quel conteneur compatible. Exemples : `std::sort`, `std::find`, `std::transform`.

ITERATORS DANS LA STL

Les itérateurs sont des objets pour parcourir les conteneurs. Ils agissent comme des pointeurs pour accéder aux éléments. Types d'itérateurs : `input_iterator`, `output_iterator`, `forward_iterator`, `bidirectional_iterator`, `random_access_iterator`. Ils permettent l'utilisation des algorithmes de la STL sur les conteneurs.

ALLOCATION DYNAMIQUE DANS LA STL

La STL utilise l'allocation dynamique pour gérer la mémoire. Les conteneurs comme `vector` et `map` allouent de la mémoire dynamiquement. Les allocateurs personnalisés peuvent être utilisés pour gérer la mémoire. Cela permet une gestion efficace et flexible de la mémoire.

PRÉSENTATION DES CONTENEURS

DÉFINITION

Les conteneurs STL sont des structures de données fournies par la bibliothèque STL. Ils permettent de stocker et manipuler des collections d'objets. Ils offrent des interfaces standardisées pour diverses opérations. Les conteneurs STL sont génériques et peuvent stocker n'importe quel type d'objet.

TYPES DE CONTENEURS

Les conteneurs STL sont classés en trois catégories principales :

- Conteneurs séquentiels
- Conteneurs associatifs
- Conteneurs adaptatifs

CONTENEURS SÉQUENTIELS

Les conteneurs séquentiels stockent les éléments dans un ordre linéaire. Les principaux conteneurs séquentiels sont :

- `vector`
- `deque`
- `list`
- `array`

CONTENEURS ASSOCIATIFS

Les conteneurs associatifs stockent les éléments en fonction de clés spécifiques. Les principaux conteneurs associatifs sont :

- map
- set
- multimap
- multiset

CONTENEURS ADAPTATIFS

Les conteneurs adaptatifs sont des conteneurs qui adaptent d'autres conteneurs. Les principaux conteneurs adaptatifs sont :

- stack
- queue
- priority_queue

COMPARAISON DES CONTENEURS

Conteneur	Accès aléatoire	Insertion/Suppression	Utilisation typique
vector	Oui	Fin	Listes dynamiques
list	Non	Partout	Listes doublement chaînées
map	Non	Clé	Dictionnaires
stack	Non	Fin	Piles

ALLOCATION DE MÉMOIRE

Les conteneurs STL utilisent des allocateurs pour gérer la mémoire. Les allocateurs permettent de personnaliser la gestion de la mémoire. Par défaut, `std::allocator` est utilisé pour l'allocation de mémoire.

UTILISATION DES ITERATORS

Les itérateurs sont des objets qui pointent vers les éléments des conteneurs. Ils permettent de parcourir les éléments de manière séquentielle. Les principaux types d'itérateurs sont :

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`

AVANTAGES

- Interfaces standardisées
- Flexibilité et réutilisabilité
- Gestion automatique de la mémoire

INCONVÉNIENTS

- Overhead de performance
- Complexité de certaines opérations
- Nécessite une compréhension approfondie pour une utilisation efficace

UTILISATION DE VECTOR

DEFINITION

`vector` est un conteneur séquentiel dynamique en C++. Il est fourni par la bibliothèque standard STL. Les éléments sont stockés de manière contiguë en mémoire. Il permet un accès rapide aux éléments via des indices. Les `vector` peuvent redimensionner automatiquement leur capacité.

DECLARATION ET INITIALISATION

```
#include <vector>

std::vector<int> vec1; // vector vide
std::vector<int> vec2(5, 10); // vector de 5 éléments, chacun initialisé à 10
std::vector<int> vec3 = {1, 2, 3, 4, 5}; // vector initialisé avec une liste
```

AJOUT D'ELEMENTS

```
std::vector<int> vec;  
vec.push_back(1); // Ajoute 1 à la fin  
vec.push_back(2); // Ajoute 2 à la fin  
vec.insert(vec.begin(), 0); // Insère 0 au début
```

ACCES AUX ELEMENTS

```
std::vector<int> vec = {1, 2, 3, 4, 5};  
int first = vec[0]; // Accès via l'opérateur []  
int second = vec.at(1); // Accès via la méthode at()
```

ITERATION AVEC ITERATOR

```
std::vector<int> vec = {1, 2, 3, 4, 5};  
for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {  
    std::cout << *it << " ";  
}
```

MODIFICATION DES ELEMENTS

```
std::vector<int> vec = {1, 2, 3, 4, 5};  
vec[0] = 10; // Modifie le premier élément  
vec.at(1) = 20; // Modifie le deuxième élément
```

SUPPRESSION DES ELEMENTS

```
std::vector<int> vec = {1, 2, 3, 4, 5};  
vec.pop_back(); // Supprime le dernier élément  
vec.erase(vec.begin()); // Supprime le premier élément  
vec.clear(); // Supprime tous les éléments
```

CAPACITE ET ALLOCATION

```
std::vector<int> vec;
vec.reserve(10); // Réserve de la mémoire pour 10 éléments
size_t size = vec.size(); // Taille actuelle
size_t capacity = vec.capacity(); // Capacité actuelle
```

AVANTAGES ET INCONVENIENTS

Avantages :

- Accès rapide par index
- Redimensionnement dynamique
- Bonne performance pour les ajouts et suppressions en fin

Inconvénients :

- Coût élevé pour les insertions/suppressions au milieu
- Peut entraîner une fragmentation de la mémoire

EXEMPLE D'UTILISATION

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    vec.push_back(6);
    vec[0] = 10;
    for (int i : vec) {
        std::cout << i << " ";
    }
    return 0;
}
```

UTILISATION DE MAP

DÉFINITION

`std::map` est un conteneur associatif de la bibliothèque STL. Il stocke des paires clé-valeur, chaque clé étant unique. Les éléments sont automatiquement triés par clé. La clé est utilisée pour accéder à la valeur associée. Les opérations de recherche, d'insertion et de suppression sont $\log(n)$.

QUAND L'UTILISER

Utilisez `std::map` lorsque :

- Vous avez besoin d'associer des valeurs à des clés uniques.
- Vous devez accéder rapidement aux valeurs par clé.
- Vous avez besoin d'un conteneur trié par clé.
- Vous effectuez fréquemment des recherches, insertions ou suppressions.

SYNTAXE

Déclaration d'une `std::map` :

```
#include <map>  
std::map<KeyType, ValueType> mapName;
```

Exemple avec des types spécifiques :

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

EXEMPLE

```
#include <map>
#include <iostream>

int main() {
    std::map<int, std::string> myMap;
    myMap[1] = "One";
    myMap[2] = "Two";
    std::cout << myMap[1] << std::endl;
    return 0;
}
```

ACCÈS AUX ÉLÉMENTS

Accès par clé :

```
std::string value = myMap[key];
```

Utilisation de **at** :

```
std::string value = myMap.at(key);
```

INSERTION D'ÉLÉMENTS

Insertion avec l'opérateur [] :

```
myMap[key] = value;
```

Insertion avec `insert` :

```
myMap.insert(std::make_pair(key, value));
```

SUPPRESSION D'ÉLÉMENTS

Suppression par clé :

```
myMap.erase(key);
```

Suppression par itérateur :

```
auto it = myMap.find(key);
if (it != myMap.end()) {
    myMap.erase(it);
}
```

ITÉRATEURS

Déclaration d'itérateur :

```
std::map<int, std::string>::iterator it;
```

Parcours avec itérateur :

```
for (it = myMap.begin(); it != myMap.end(); ++it) {
    std::cout << it->first << " : " << it->second << std::endl;
}
```

AVANTAGES ET INCONVÉNIENTS

Avantages :

- Accès rapide aux éléments par clé.
- Tri automatique des clés.
- Recherche, insertion, suppression en $\log(n)$.

Inconvénients :

- Utilise plus de mémoire que des conteneurs séquentiels.
- Plus lent pour l'accès séquentiel par rapport à `std::vector`.

ALLOCATION ET GESTION DE LA MÉMOIRE

`std::map` utilise un arbre binaire de recherche équilibré. La mémoire est allouée dynamiquement pour les nœuds de l'arbre. Les allocations sont gérées automatiquement par le conteneur. Les éléments sont stockés de manière non contiguë en mémoire. La gestion de la mémoire est optimisée pour les opérations de recherche et de tri.

UTILISATION DE STACK

DEFINITION

Une stack (pile) est une structure de données LIFO (Last In, First Out). Les éléments sont ajoutés et retirés du sommet de la pile. Utilisée pour des opérations où l'ordre d'accès est crucial. Exemples : gestion d'appels de fonctions, annulation d'actions.

QUAND L'UTILISER

Utiliser une stack lorsque :

- Vous avez besoin d'un accès LIFO.
- La gestion de l'état est cruciale (par ex. pile d'appels).
- Vous devez annuler des opérations (par ex. undo).

SYNTAXE

Inclure la bibliothèque <stack> :

```
#include <stack>
std::stack<int> myStack;
```

EXAMPLE

```
#include <stack>
#include <iostream>

int main() {
    std::stack<int> myStack;
    myStack.push(10);
    myStack.push(20);
    std::cout << "Top element: " << myStack.top() << std::endl;
    return 0;
}
```

OPERATIONS BASIQUES (PUSH, POP, TOP)

Opération Syntaxe

Push	<code>myStack.push(value);</code>
------	-----------------------------------

Pop	<code>myStack.pop();</code>
-----	-----------------------------

Top	<code>myStack.top();</code>
-----	-----------------------------

ITERATION AVEC STACK

Les stacks ne supportent pas l'itération directe. Pour itérer, vider la stack :

```
while (!myStack.empty()) {  
    std::cout << myStack.top() << std::endl;  
    myStack.pop();  
}
```

ALLOCATION DE MÉMOIRE

Les stacks allouent de la mémoire dynamiquement. La taille de la stack peut croître ou diminuer. Pas de gestion manuelle de la mémoire nécessaire.

AVANTAGES ET INCONVENIENTS

Avantages :

- Simple à utiliser.
- Gestion automatique de la mémoire.

Inconvénients :

- Accès limité (seulement au sommet).
- Pas d'itération directe possible.

UTILISATION DE QUEUE

DEFINITION

Une queue (file) est une structure de données linéaire. Elle suit le principe FIFO (First In, First Out). Le premier élément inséré est le premier à être retiré. Elle est utilisée pour gérer les processus en attente. La bibliothèque STL de C++ fournit une implémentation de queue.

QUAND L'UTILISER

Utilisez une queue lorsque :

- Vous avez besoin de traiter des éléments dans l'ordre d'arrivée.
- Vous gérez des tâches en attente ou des processus.
- Vous implémentez des algorithmes de parcours en largeur (BFS).
- Vous gérez des systèmes de messagerie ou des files d'attente de tâches.

SYNTAXE

Pour utiliser une queue en C++, incluez la bibliothèque `<queue>`:

```
#include <queue>
std::queue<int> myQueue;
```

EXEMPLE

Voici un exemple d'utilisation de queue :

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> myQueue;
    myQueue.push(10);
    myQueue.push(20);
    myQueue.push(30);
    std::cout << "Front: " << myQueue.front() << std::endl;
    myQueue.pop();
    std::cout << "Front after pop: " << myQueue.front() << std::endl;
    return 0;
}
```

OPERATIONS PRINCIPALES

Opération	Description	Syntaxe
Enfiler (push)	Ajouter un élément à la fin	<code>myQueue.push(x)</code>
Défiler (pop)	Retirer l'élément en tête	<code>myQueue.pop()</code>
Tête (front)	Accéder à l'élément en tête	<code>myQueue.front()</code>
Taille (size)	Obtenir le nombre d'éléments	<code>myQueue.size()</code>
Vide (empty)	Vérifier si la queue est vide	<code>myQueue.empty()</code>

AVANTAGES

- Simple à utiliser.
- Gère efficacement les processus en attente.
- FIFO garantit l'ordre de traitement.

INCONVÉNIENTS

- Accès limité (seule la tête est accessible).
- Pas de support pour l'itération directe.

COMPARAISON AVEC STACK

Caractéristique	Queue (file)	Stack (pile)
Principe	FIFO (First In, First Out)	LIFO (Last In, First Out)
Ajout d'élément	push à la fin	push au sommet
Retrait d'élément	pop de la tête	pop du sommet
Accès	front pour la tête	top pour le sommet

ALLOCATION DES CONTENEURS

Les queues peuvent être implémentées avec différents conteneurs :

- `std::deque` (par défaut)
- `std::list`

```
std::queue<int, std::deque<int>> myQueue1;
std::queue<int, std::list<int>> myQueue2;
```

UTILISATION AVEC ITERATOR

Les queues n'ont pas d'itérateurs directs. Cependant, vous pouvez transférer les éléments vers un autre conteneur. Utilisez un `std::vector` ou `std::deque` pour itérer :

```
std::queue<int> myQueue;
// Transférer les éléments dans un vector
std::vector<int> myVector;
while (!myQueue.empty()) {
    myVector.push_back(myQueue.front());
    myQueue.pop();
}
// Itérer sur le vector
for (int elem : myVector) {
    std::cout << elem << std::endl;
}
```

ITERATORS

DÉFINITION

Les itérateurs sont des objets qui permettent de parcourir des éléments d'un conteneur. Ils agissent comme des pointeurs pour accéder aux éléments. Les itérateurs sont utilisés pour traverser des structures de données. Ils sont une abstraction importante en C++ pour la manipulation des conteneurs STL.

TYPES D'ITÉRATEURS

- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator

ITÉRATEURS AVEC MAP

```
std::map<int, std::string> myMap = {{1, "one"}, {2, "two"}};
for (auto it = myMap.begin(); it != myMap.end(); ++it) {
    std::cout << it->first << " : " << it->second << "\n";
}
```

ITÉRATEURS AVEC STACK

Les stacks ne supportent pas directement les itérateurs. Utilisez des méthodes comme `top()`, `push()`, et `pop()` pour manipuler les éléments. Pour parcourir une stack, il faut souvent la vider.

ITÉRATEURS AVEC QUEUE

Les queues ne supportent pas directement les itérateurs. Utilisez des méthodes comme `front()`, `back()`, `push()`, et `pop()` pour manipuler les éléments. Pour parcourir une queue, il faut souvent la vider.

AVANTAGES DES ITÉRATEURS

- Abstraction des détails de la structure de données
- Uniformité d'accès aux éléments
- Facilite la manipulation et la traversée des conteneurs
- Permet l'utilisation d'algorithmes génériques

BONNES PRATIQUES

- Toujours vérifier si l'itérateur est valide avant de l'utiliser.
- Utiliser `const_iterator` pour les conteneurs non modifiables.
- Préférer les boucles `for` basées sur les itérateurs pour une meilleure lisibilité.
- Éviter de modifier le conteneur lors de l'itération.

ERREURS COURANTES

- Déréférencement d'un itérateur invalide.
- Modification du conteneur pendant l'itération.
- Utilisation incorrecte des types d'itérateurs.
- Confusion entre les différents types d'itérateurs (e.g., `begin()` vs `cbegin()`).

ALLOCATION DYNAMIQUE DES CONTENEURS

DÉFINITION

L'allocation dynamique permet de réserver de la mémoire pendant l'exécution du programme. Les conteneurs de la STL utilisent l'allocation dynamique pour gérer leur taille. Cela permet aux conteneurs de s'adapter dynamiquement à la quantité de données.

QUAND L'UTILISER

Utilisez l'allocation dynamique lorsque la taille des données n'est pas connue à l'avance. Elle est utile pour les structures de données qui changent de taille fréquemment. Elle permet une gestion efficace de la mémoire en évitant le gaspillage.

SYNTAXE

Pour la plupart des conteneurs STL, l'allocation dynamique est gérée automatiquement. Par exemple, pour un **vector**, la syntaxe est :

```
std::vector<int> v;
```

EXEMPLE

```
std::vector<int> v;  
v.push_back(1);  
v.push_back(2);  
v.push_back(3);
```

Ici, le `vector` s'agrandit dynamiquement pour accueillir les nouveaux éléments.

ALLOCATION DYNAMIQUE AVEC VECTOR

Le `vector` utilise un tableau dynamique pour stocker ses éléments. Il double généralement sa capacité lorsqu'il atteint sa limite. Cela minimise les réallocations et optimise les performances.

ALLOCATION DYNAMIQUE AVEC MAP

`std::map` utilise des arbres binaires pour stocker les paires clé-valeur. L'allocation dynamique permet de gérer l'ajout et la suppression de paires. Elle assure une complexité logarithmique pour les opérations de recherche.

ALLOCATION DYNAMIQUE AVEC STACK

`std::stack` est une adaptation de conteneur, souvent basée sur `std::deque`. L'allocation dynamique permet de gérer la pile de manière flexible. Elle permet d'ajouter et de retirer des éléments sans se soucier de la taille initiale.

ALLOCATION DYNAMIQUE AVEC QUEUE

`std::queue` est aussi une adaptation de conteneur, souvent basée sur `std::deque`. L'allocation dynamique permet de gérer la file d'attente efficacement. Elle permet d'ajouter des éléments à l'arrière et de les retirer à l'avant.

AVANTAGES ET INCONVÉNIENTS

Avantages :

- Flexibilité dans la gestion de la mémoire.
- Adaptation dynamique à la taille des données.
- Réduction du gaspillage de mémoire.

Inconvénients :

- Complexité de gestion de la mémoire.
- Risque de fragmentation de la mémoire.
- Potentiel de surcoût en performance lors des réallocations.

GESTION DE LA MÉMOIRE AVEC LES CONTENEURS

INTRODUCTION À LA GESTION DE LA MÉMOIRE

La gestion de la mémoire est cruciale pour optimiser les performances. Les conteneurs STL facilitent la gestion de la mémoire. Ils allouent et libèrent la mémoire dynamiquement. Cela réduit les erreurs de mémoire courantes comme les fuites. Comprendre comment chaque conteneur gère la mémoire est essentiel. Cela aide à choisir le conteneur le plus approprié. Optimiser la gestion de la mémoire améliore l'efficacité du programme.

ALLOCATION DYNAMIQUE ET CONTENEURS

Les conteneurs STL utilisent l'allocation dynamique. L'allocation dynamique permet de gérer la mémoire à l'exécution. Les conteneurs comme `vector` et `map` gèrent automatiquement la mémoire. Ils utilisent des allocateurs pour gérer la mémoire. Les allocateurs allouent et libèrent la mémoire selon les besoins. Cela permet une utilisation efficace de la mémoire.

GESTION DE LA MÉMOIRE AVEC VECTOR

`vector` utilise un tableau dynamique pour stocker les éléments. Il alloue de la mémoire supplémentaire pour anticiper les ajouts. La mémoire est réallouée lorsque la capacité est dépassée. La réallocation peut entraîner des copies d'éléments. La méthode `shrink_to_fit` réduit la capacité à la taille actuelle. Utiliser `reserve` pour allouer de la mémoire à l'avance.

GESTION DE LA MÉMOIRE AVEC MAP

`map` utilise une structure d'arbre pour stocker les éléments. Chaque élément est un nœud dans l'arbre. La mémoire est allouée pour chaque nœud inséré. Les nœuds sont reliés par des pointeurs. La suppression d'un élément libère la mémoire du nœud. Les itérateurs sont invalidés après la suppression.

GESTION DE LA MÉMOIRE AVEC STACK

`stack` est un adaptateur de conteneur. Il utilise un conteneur sous-jacent comme `deque` ou `vector`. La gestion de la mémoire dépend du conteneur sous-jacent. Les opérations `push` et `pop` allouent et libèrent la mémoire. La mémoire est gérée automatiquement par le conteneur sous-jacent.

GESTION DE LA MÉMOIRE AVEC QUEUE

`queue` est un adaptateur de conteneur. Il utilise un conteneur sous-jacent comme `deque` ou `list`. La gestion de la mémoire dépend du conteneur sous-jacent. Les opérations `push` et `pop` allouent et libèrent la mémoire. La mémoire est gérée automatiquement par le conteneur sous-jacent.

UTILISATION D'ITERATOR ET GESTION DE LA MÉMOIRE

Les itérateurs parcourent les éléments des conteneurs. Ils ne gèrent pas directement la mémoire. La suppression d'éléments peut invalider les itérateurs. Utiliser des itérateurs valides pour éviter des erreurs. Les itérateurs facilitent l'accès et la manipulation des éléments.

BONNES PRATIQUES DE GESTION DE LA MÉMOIRE

Prévoir la capacité des conteneurs avec `reserve`. Utiliser des méthodes comme `clear` pour libérer la mémoire. Éviter les copies inutiles d'éléments. Utiliser des itérateurs pour manipuler les éléments efficacement. Surveiller l'utilisation de la mémoire pour optimiser les performances.

ERREURS COURANTES ET SOLUTIONS

Fuites de mémoire : utiliser des conteneurs pour une gestion automatique. Itérateurs invalides : vérifier la validité des itérateurs après modification. Réallocations fréquentes : utiliser `reserve` pour allouer la mémoire à l'avance. Utilisation inefficace de la mémoire : choisir le conteneur approprié.

COMPARAISON DES CONTENEURS

DEFINITION

Les conteneurs standard de la bibliothèque STL en C++ incluent **vector**, **map**, **stack**, et **queue**. Chaque conteneur a des caractéristiques et des usages spécifiques. **vector** est une séquence dynamique de taille variable. **map** est une collection d'éléments clé-valeur, unique par clé. **stack** est une structure de données LIFO (Last-In-First-Out). **queue** est une structure de données FIFO (First-In-First-Out).

CRITÈRES DE COMPARAISON

Les conteneurs peuvent être comparés selon :

- Performance
- Utilisation de la mémoire
- Simplicité d'utilisation
- Utilisation d'iterator
- Allocation dynamique
- Cas d'utilisation typiques
- Avantages et inconvénients

PERFORMANCE

`vector` a un accès aléatoire en temps constant ($O(1)$). `map` a un accès en temps logarithmique ($O(\log n)$). `stack` et `queue` ont des opérations de base en temps constant ($O(1)$). Les performances dépendent de l'usage spécifique et de la taille des données.

UTILISATION DE LA MÉMOIRE

`vector` utilise de la mémoire contiguë, ce qui peut entraîner des reallocations. `map` utilise une structure d'arbre, consommant plus de mémoire pour les pointeurs. `stack` et `queue` utilisent des listes ou des vecteurs sous-jacents. La gestion de la mémoire varie selon le conteneur et l'implémentation.

SIMPLICITÉ D'UTILISATION

`vector` est simple à utiliser pour des séquences dynamiques. `map` est plus complexe en raison des paires clé-valeur. `stack` et `queue` offrent des interfaces simples pour les opérations LIFO et FIFO. Le choix du conteneur dépend de la simplicité requise par l'application.

UTILISATION D'ITERATOR

Tous les conteneurs STL supportent les itérateurs. `vector` a des itérateurs pour un accès séquentiel. `map` a des itérateurs pour parcourir les paires clé-valeur. `stack` et `queue` n'ont pas d'itérateurs explicites, mais peuvent être parcourus via leurs conteneurs sous-jacents.

ALLOCATION DYNAMIQUE

`vector` utilise l'allocation dynamique et peut redimensionner automatiquement. `map` alloue dynamiquement des nœuds pour les paires clé-valeur. `stack` et `queue` dépendent de leurs conteneurs sous-jacents pour l'allocation. La gestion de la mémoire est automatique pour tous les conteneurs STL.

CAS D'UTILISATION TYPIQUES

`vector` est utilisé pour des listes dynamiques de taille variable. `map` est utilisé pour des associations clé-valeur, comme des dictionnaires. `stack` est utilisé pour des algorithmes de pile, comme le parcours en profondeur. `queue` est utilisé pour des algorithmes de file, comme le parcours en largeur.

AVANTAGES ET INCONVÉNIENTS

`vector` :

- Avantages : Accès rapide, simple à utiliser.
- Inconvénients : Reallocations coûteuses.

`map` :

- Avantages : Accès rapide par clé, éléments uniques.
- Inconvénients : Plus de mémoire utilisée.

`stack` :

- Avantages : Simple pour LIFO.
- Inconvénients : Accès limité.

`queue` :

- Avantages : Simple pour FIFO.
- Inconvénients : Accès limité.

MÉTHODES ET OPÉRATIONS DE BASE SUR VECTOR

DEFINITION

`vector` est un conteneur séquentiel de la bibliothèque STL en C++. Il fonctionne comme un tableau dynamique. Permet l'accès direct aux éléments via des indices. Les éléments sont stockés de manière contiguë en mémoire. La taille peut changer dynamiquement avec l'ajout ou la suppression d'éléments. Offre des méthodes pour gérer la taille et la capacité.

QUAND L'UTILISER

Utilisez `vector` lorsque :

- Vous avez besoin d'accéder aux éléments par index.
- La taille de la collection peut changer fréquemment.
- Vous avez besoin de performance proche de celle des tableaux.
- Vous souhaitez utiliser des fonctionnalités avancées de la STL.

AJOUTER DES ELEMENTS

```
std::vector<int> vec;  
vec.push_back(10); // Ajoute 10 à la fin  
vec.insert(vec.begin(), 5); // Insère 5 au début
```

ACCÉDER AUX ELEMENTS

```
std::vector<int> vec = {1, 2, 3};  
int first = vec[0]; // Accès par index  
int second = vec.at(1); // Accès sécurisé par at()
```

MODIFIER DES ELEMENTS

```
std::vector<int> vec = {1, 2, 3};  
vec[0] = 10; // Modification par index  
vec.at(1) = 20; // Modification sécurisée par at()
```

SUPPRIMER DES ELEMENTS

```
std::vector<int> vec = {1, 2, 3};  
vec.pop_back(); // Supprime le dernier élément  
vec.erase(vec.begin()); // Supprime le premier élément
```

ITERATION AVEC ITERATOR

```
std::vector<int> vec = {1, 2, 3};  
for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {  
    std::cout << *it << " ";  
}
```

ALLOCATION DE MEMOIRE

```
std::vector<int> vec;  
vec.reserve(100); // Réserve de la mémoire pour 100 éléments  
vec.resize(50); // Redimensionne le vecteur à 50 éléments
```

AVANTAGES ET INCONVENIENTS

Avantages :

- Accès rapide par index.
- Taille dynamique.
- Mémoire contiguë.

Inconvénients :

- Coût de réallocation mémoire.
- Moins efficace pour les insertions/suppressions au milieu.

MÉTHODES ET OPÉRATIONS DE BASE SUR MAP

DEFINITION

Un `map` en C++ est un conteneur associatif. Il stocke des paires clé-valeur. Chaque clé est unique et est associée à une valeur. Les clés sont ordonnées selon un ordre strict. Les opérations de recherche, insertion et suppression sont rapides. Les clés doivent être comparables.

QUAND L'UTILISER

Utilisez un `map` lorsque :

- Vous avez besoin d'associer des clés uniques à des valeurs.
- Vous devez effectuer des recherches rapides par clé.
- Vous avez besoin d'un conteneur ordonné.
- Vous voulez garantir l'unicité des clés.

SYNTAXE

La syntaxe de déclaration d'un `map` est la suivante :

```
#include <map>
std::map<KeyType, ValueType> mapName;
```

Exemple :

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

INSERTION D'ÉLÉMENTS

Pour insérer des éléments dans un map :

```
myMap.insert(std::make_pair(key, value));  
myMap[key] = value;
```

Exemple :

```
myMap.insert(std::make_pair(1, "One"));  
myMap[2] = "Two";
```

ACCÈS AUX ÉLÉMENTS

Pour accéder aux éléments d'un map :

```
value = myMap[key];
```

Exemple :

```
std::string value = myMap[1]; // "One"
```

SUPPRESSION D'ÉLÉMENTS

Pour supprimer des éléments d'un `map` :

```
myMap.erase(key);
```

Exemple :

```
myMap.erase(1); // Supprime l'élément avec la clé 1
```

ITERATION AVEC ITERATOR

Pour itérer sur les éléments d'un map :

```
for (auto it = myMap.begin(); it != myMap.end(); ++it) {
    std::cout << it->first << " : " << it->second << std::endl;
}
```

RECHERCHE D'ÉLÉMENTS

Pour rechercher des éléments dans un `map` :

```
auto it = myMap.find(key);
if (it != myMap.end()) {
    // Clé trouvée, it->second contient la valeur
}
```

ALLOCATION ET GESTION DE MÉMOIRE

Un `map` gère automatiquement la mémoire pour ses éléments. Les éléments sont alloués dynamiquement. La mémoire est libérée lorsque les éléments sont supprimés. L'utilisation de `map` peut augmenter la fragmentation de la mémoire.

EXEMPLE D'UTILISATION

```
#include <iostream>
#include <map>

int main() {
    std::map<int, std::string> myMap;
    myMap[1] = "One";
    myMap[2] = "Two";

    for (const auto& pair : myMap) {
        std::cout << pair.first << " : " << pair.second << std::endl;
    }

    return 0;
}
```

AVANTAGES ET INCONVÉNIENTS

Avantages :

- Accès rapide par clé.
- Clés uniques et ordonnées.
- Bonne performance pour les opérations de recherche.

Inconvénients :

- Consommation mémoire plus élevée.
- Moins efficace pour les petites collections.
- Complexité de la gestion des clés comparables.

MÉTHODES ET OPÉRATIONS DE BASE SUR STACK

DEFINITION

Une stack (pile) est une structure de données LIFO (Last In, First Out). Les éléments sont ajoutés et retirés du même côté. Elle est utilisée pour des opérations où le dernier élément ajouté doit être le premier retiré.

QUAND L'UTILISER

- Algorithmes de parcours (DFS)
- Gestion des appels récursifs
- Annulation d'opérations (undo)
- Évaluation d'expressions (parsing)

EXEMPLE D'USAGE

```
#include <stack>
#include <iostream>

int main() {
    std::stack<int> myStack;
    myStack.push(10);
    myStack.push(20);
    std::cout << myStack.top(); // Affiche 20
    return 0;
}
```

PUSH()

Ajoute un élément au sommet de la stack.

```
myStack.push(30);
```

POP()

Retire l'élément au sommet de la stack.

```
myStack.pop();
```

TOP()

Accède à l'élément au sommet de la stack sans le retirer.

```
int topElement = myStack.top();
```

EMPTY()

Vérifie si la stack est vide.

```
bool isEmpty = myStack.empty();
```

SIZE()

Retourne le nombre d'éléments dans la stack.

```
size_t size = myStack.size();
```

UTILISATION D'ITERATOR AVEC STACK

Les stacks n'ont pas d'itérateurs car elles sont basées sur LIFO. Pour parcourir, utilisez pop() et top().

```
while (!myStack.empty()) {  
    std::cout << myStack.top() << " ";  
    myStack.pop();  
}
```

ALLOCATION DE STACK

Les stacks peuvent être allouées dynamiquement.

```
std::stack<int>* dynamicStack = new std::stack<int>();
delete dynamicStack;
```

MÉTHODES ET OPÉRATIONS DE BASE SUR QUEUE

DEFINITION

Une queue (file) est un conteneur de type FIFO (First In, First Out). Les éléments sont ajoutés à l'arrière et retirés à l'avant. Elle est utile pour gérer des tâches dans l'ordre de leur arrivée. La bibliothèque STL de C++ fournit la classe `queue` pour ce conteneur. Elle est incluse dans l'en-tête `<queue>`.

QUAND L'UTILISER

Utilisez une queue lorsque vous avez besoin de traiter des éléments dans l'ordre de leur arrivée. Exemples d'utilisation :

- Gestion des tâches dans un système d'exploitation.
- Traitement des requêtes dans un serveur web.
- Simulation de files d'attente dans des systèmes de service.

SYNTAXE

Pour utiliser une queue en C++, incluez l'en-tête `<queue>` et déclarez une queue comme suit :

```
#include <queue>  
std::queue<int> q;
```

EXEMPLE

Voici un exemple de base de l'utilisation d'une queue :

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(10);
    q.push(20);
    q.push(30);
    std::cout << "Front: " << q.front() << std::endl;
    std::cout << "Back: " << q.back() << std::endl;
    return 0;
}
```

PUSH

La méthode `push` ajoute un élément à l'arrière de la queue.

Syntaxe :

```
q.push(value);
```

Exemple :

```
q.push(10);
```

POP

La méthode `pop` retire l'élément à l'avant de la queue.

Syntaxe :

```
q.pop();
```

Exemple :

```
q.pop();
```

FRONT

La méthode `front` retourne une référence à l'élément situé à l'avant de la queue.

Syntaxe :

```
q.front();
```

Exemple :

```
int frontElement = q.front();
```

BACK

La méthode `back` retourne une référence à l'élément situé à l'arrière de la queue.

Syntaxe :

```
q.back();
```

Exemple :

```
int backElement = q.back();
```

EMPTY

La méthode `empty` vérifie si la queue est vide.

Syntaxe :

```
q.empty();
```

Exemple :

```
if (q.empty()) {
    std::cout << "Queue is empty." << std::endl;
}
```

SIZE

La méthode `size` retourne le nombre d'éléments dans la queue.

Syntaxe :

```
q.size();
```

Exemple :

```
size_t queueSize = q.size();
```


AVANTAGES :

- Simple à utiliser pour des opérations FIFO.
- Efficace pour gérer des tâches dans l'ordre d'arrivée.

INCONVÉNIENTS :

- Accès limité aux éléments (seulement front et back).
- Moins flexible que d'autres conteneurs comme `vector` ou `deque`.

PARCOURS DES CONTENEURS AVEC ITERATORS

DEFINITION DES ITERATORS

Les iterators sont des objets qui pointent vers les éléments d'un conteneur. Ils permettent de parcourir les éléments d'un conteneur de manière séquentielle. Les iterators sont similaires aux pointeurs en C++. Ils supportent les opérations d'incrémentation et de décrémentation. Ils sont utilisés pour accéder et manipuler les éléments des conteneurs STL.

TYPES D'ITERATORS

Il existe plusieurs types d'iterators en C++ :

- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator

SYNTAXE DE BASE

La syntaxe de base pour déclarer un iterator est la suivante :

```
std::vector<int>::iterator it;
```

Pour initialiser un iterator, on utilise des méthodes comme `begin()` et `end()` :

```
it = vec.begin();
```

PARCOURS AVEC ITERATORS

Pour parcourir un conteneur avec un iterator, on utilise une boucle :

```
for (auto it = vec.begin(); it != vec.end(); ++it) {  
    std::cout << *it << " ";  
}
```

L'iterator `it` est incrémenté à chaque itération de la boucle.

ITERATORS AVEC VECTOR

Les vectors supportent les iterators :

```
std::vector<int> vec = {1, 2, 3, 4};  
for (auto it = vec.begin(); it != vec.end(); ++it) {  
    std::cout << *it << " ";  
}
```

Les iterators pour `vector` sont de type Random Access Iterator.

ITERATORS AVEC MAP

Les maps utilisent des iterators pour parcourir les paires clé-valeur :

```
std::map<int, std::string> m = {{1, "one"}, {2, "two"}};
for (auto it = m.begin(); it != m.end(); ++it) {
    std::cout << it->first << ": " << it->second << "\n";
}
```

Les iterators pour map sont de type Bidirectional Iterator.

ITERATORS AVEC STACK

Les stacks ne supportent pas directement les iterators. Pour parcourir une stack, il faut utiliser des méthodes comme `top()` et `pop()` :

```
std::stack<int> s;
while (!s.empty()) {
    std::cout << s.top() << " ";
    s.pop();
}
```

ITERATORS AVEC QUEUE

Les queues ne supportent pas directement les iterators. Pour parcourir une queue, on utilise des méthodes comme `front()` et `pop()` :

```
std::queue<int> q;
while (!q.empty()) {
    std::cout << q.front() << " ";
    q.pop();
}
```

AVANTAGES DES ITERATORS

- Abstraction du parcours des conteneurs.
- Uniformité : même interface pour différents conteneurs.
- Flexibilité : possibilité de manipuler les éléments.
- Compatibilité avec les algorithmes STL.

LIMITATIONS DES ITERATORS

- Pas toujours intuitifs pour les débutants.
- Certaines opérations peuvent être moins performantes.
- Les iterators invalides peuvent causer des erreurs.
- Pas supportés par tous les conteneurs (ex. stack, queue).

UTILISATION D'ALGORITHMES STL AVEC LES CONTENEURS

DÉFINITION DES ALGORITHMES STL

Les algorithmes STL sont des fonctions génériques. Ils opèrent sur des conteneurs via des iterators. Ils permettent des opérations comme la recherche, le tri, la modification, etc. Ils sont définis dans l'en-tête `<algorithm>`. Ils sont indépendants des types de conteneurs. Ils augmentent la réutilisabilité et la maintenabilité du code.

ALGORITHMES DE RECHERCHE

Les algorithmes de recherche permettent de trouver des éléments. Exemples :

- `std::find` : Trouve la première occurrence d'un élément.
- `std::binary_search` : Vérifie si un élément existe (prérequis : conteneur trié).
- `std::find_if` : Trouve le premier élément qui satisfait une condition.

ALGORITHMES DE TRI

Les algorithmes de tri réorganisent les éléments dans un ordre spécifique. Exemples :

- `std::sort` : Trie les éléments dans l'ordre croissant par défaut.
- `std::stable_sort` : Trie tout en préservant l'ordre relatif des éléments égaux.
- `std::partial_sort` : Trie partiellement un conteneur.

ALGORITHMES DE MODIFICATION

Les algorithmes de modification changent les éléments des conteneurs. Exemples :

- `std::copy` : Copie des éléments d'un conteneur à un autre.
- `std::transform` : Applique une fonction à chaque élément.
- `std::replace` : Remplace les occurrences d'un élément par un autre.

ALGORITHMES DE PARTITIONNEMENT

Les algorithmes de partitionnement réorganisent les éléments selon une condition. Exemples :

- `std::partition` : Réorganise les éléments pour que ceux qui satisfont une condition soient en premier.
- `std::stable_partition` : Comme `std::partition` mais préserve l'ordre relatif des éléments.
- `std::partition_copy` : Copie les éléments en deux groupes selon une condition.

ALGORITHMES DE FUSION

Les algorithmes de fusion combinent des conteneurs triés. Exemples :

- `std::merge` : Fusionne deux conteneurs triés en un seul.
- `std::inplace_merge` : Fusionne deux parties triées d'un conteneur.
- `std::set_union` : Fusionne deux ensembles triés en un ensemble trié unique.

UTILISATION D'ITERATORS AVEC LES ALGORITHMES

Les iterators sont utilisés pour spécifier les plages d'éléments. Les algorithmes STL prennent souvent des iterators en arguments. Exemples :

- `std::vector<int>::iterator it = std::find(vec.begin(), vec.end(), 5);`
- `std::sort(vec.begin(), vec.end());` Les iterators peuvent être des pointeurs ou des objets plus complexes.

ALLOCATION DYNAMIQUE ET ALGORITHMES

Les algorithmes STL peuvent être utilisés avec des conteneurs dynamiquement alloués. Exemples :

- `std::vector<int> *vec = new std::vector<int>({1, 2, 3});`
- `std::sort(vec->begin(), vec->end());` Attention à la gestion de la mémoire lors de l'utilisation dynamique.

EXEMPLES PRATIQUES

```
#include <algorithm>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> vec = {3, 1, 4, 1, 5, 9};
    std::sort(vec.begin(), vec.end());
    for (int n : vec) std::cout << n << " ";
    return 0;
}
```


AVANTAGES

- Réutilisabilité du code.
- Performance optimisée.
- Simplification du code.

LIMITATIONS

- Nécessite une compréhension des iterators.
- Moins flexible que des implémentations spécifiques.
- Peut nécessiter des conteneurs triés pour certains algorithmes.

CONVERSION ENTRE DIFFÉRENTS TYPES DE CONTENEURS

DEFINITION

La conversion entre différents types de conteneurs STL consiste à transférer les éléments d'un conteneur à un autre. Elle permet de bénéficier des avantages spécifiques de chaque conteneur. Les conteneurs courants incluent `vector`, `list`, `deque`, et `map`.

POURQUOI CONVERTIR ENTRE CONTENEURS

- Optimisation des performances.
- Utilisation des fonctionnalités spécifiques à chaque conteneur.
- Simplification du code.
- Facilitation de l'intégration avec d'autres bibliothèques ou API.

SYNTAXE GÉNÉRALE DE CONVERSION

La conversion entre conteneurs se fait généralement en utilisant des iterators. Exemple de syntaxe générale :

```
std::vector<int> vec = {1, 2, 3};  
std::list<int> lst(vec.begin(), vec.end());
```

CONVERSION DE VECTOR À LIST

Pour convertir un vector en list :

```
std::vector<int> vec = {1, 2, 3};  
std::list<int> lst(vec.begin(), vec.end());
```

CONVERSION DE LIST À VECTOR

Pour convertir une `list` en `vector`:

```
std::list<int> lst = {1, 2, 3};  
std::vector<int> vec(lst.begin(), lst.end());
```

CONVERSION DE VECTOR À DEQUE

Pour convertir un `vector` en `deque` :

```
std::vector<int> vec = {1, 2, 3};  
std::deque<int> deq(vec.begin(), vec.end());
```

CONVERSION DE DEQUE À VECTOR

Pour convertir une deque en vector :

```
std::deque<int> deq = {1, 2, 3};  
std::vector<int> vec(deq.begin(), deq.end());
```

CONVERSION DE MAP À VECTOR

Pour convertir un `map` en `vector`:

```
std::map<int, int> mp = {{1, 10}, {2, 20}, {3, 30}};  
std::vector<std::pair<int, int>> vec(mp.begin(), mp.end());
```

CONVERSION DE VECTOR À MAP

Pour convertir un vector en map :

```
std::vector<std::pair<int, int>> vec = {{1, 10}, {2, 20}, {3, 30}};  
std::map<int, int> mp(vec.begin(), vec.end());
```

UTILISATION DES ITERATORS POUR LA CONVERSION

Les iterators permettent de parcourir les éléments des conteneurs. Ils sont utilisés pour initialiser un nouveau conteneur à partir d'un autre. Syntaxe :

```
std::vector<int> vec = {1, 2, 3};  
std::list<int> lst(vec.begin(), vec.end());
```

ALLOCATION MÉMOIRE LORS DE LA CONVERSION

Lors de la conversion, le nouvel espace mémoire est alloué pour le conteneur cible. Il est important de tenir compte de la gestion de la mémoire pour éviter les fuites. Utiliser des méthodes appropriées pour libérer la mémoire si nécessaire.

EXEMPLES PRATIQUES DE CONVERSION

Conversion de `vector` en `list`:

```
std::vector<int> vec = {1, 2, 3};  
std::list<int> lst(vec.begin(), vec.end());
```

Conversion de `map` en `vector`:

```
std::map<int, int> mp = {{1, 10}, {2, 20}, {3, 30}};  
std::vector<std::pair<int, int>> vec(mp.begin(), mp.end());
```