Les Vecteurs

Les vecteurs en C++

Introduction aux vecteurs

Définition

Les **vecteurs** sont des conteneurs de données similaires aux **tableaux**, mais avec des **fonctionnalités avancées** et une **redimensionnement dynamique**.

Les vecteurs sont des objets de la bibliothèque standard de C++ (STL). Ils sont plus flexibles que les tableaux classiques car ils peuvent être redimensionnés facilement et offrent certaines fonctions utiles.

Avantages par rapport aux tableaux

- **Taille modifiable** à l'exécution
- Gestion de la **mémoire automatique**
- Nombreuses **fonctions intégrées**

Inclusion de la bibliothèque vector

Pour utiliser les **vecteurs** en C++, il faut inclure la **bibliothèque** correspondante avec cette ligne de code :

```
#include <vector>
```

Les vecteurs sont des conteneurs de la bibliothèque standard qui permettent de stocker des éléments de manière dynamique.

Création de vecteurs

Syntaxe de déclaration

```
std::vector<type de données> nom_du_vecteur;
```

Initialisation des vecteurs

Création d'un vecteur vide

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

Un vecteur vide permet de démontrer la syntaxe de base pour créer un vecteur, sans lui donner de contenu initial. Les éléments pourront être ajoutés plus tard.

Avec taille prédéfinie

```
std::vector<int> vecteur_de_taille(5);
```

Avec des éléments spécifiques

```
std::vector<int> vecteur_avec_elements{1, 2, 3, 4, 5};
```

Accès aux éléments d'un vecteur

Accès par index

Pour accéder aux éléments d'un **vecteur**, on utilise les crochets `[]` et l'**index** de l'élément souhaité (indexation à partir de 0).

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

int main() {
  std::vector<int> myVector = {1, 2, 3, 4, 5};

  std::cout << "L'élément à l'indice 2 est : " << myVector[2] << std::endl;
```

```
  return 0;
}
```

Utiliser des index inexistants peut causer des erreurs d'exécution. Pensez à vérifier la taille du vecteur avant d'accéder à un index.

Syntaxe

```
vector<int> my_vector = {3, 5, 7};
int val = my_vector[1]; // résultat : val = 5
```

Ici, nous créons un **vector<int>** avec des valeurs initiales et accédons à la deuxième valeur en utilisant l'index 1.

Exemples d'utilisation

```
my_vector[0] = 10; // le premier élément est maintenant 10
int somme = my_vector[0] + my_vector[1]; // somme = 15
```

Accès aux premiers et derniers éléments

Les méthodes `front()` et `back()` permettent d'accéder respectivement au **premier** et au **dernier** élément du vecteur.

Méthode	Description
front()	Accède au premier élément
back()	Accède au dernier élément

Exemple d'utilisation :

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

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};

    std::cout << "Premier élément: " << v.front() << std::endl;
    std::cout << "Dernier élément: " << v.back() << std::endl;

    return 0;
}
```

Les méthodes `front()` et `back()`

```
int first_element = my_vector.front(); // first_element = 10
int last_element = my_vector.back(); // last_element = 7
```

Exemples d'utilisation

```
my_vector.front() = 1; // le premier élément est maintenant 1
my_vector.back() = 9; // le dernier élément est maintenant 9
```

Ces fonctions permettent de modifier directement les éléments d'un vecteur en C++ sans avoir à utiliser une boucle ou un indice.

Modification de la taille d'un vecteur

Méthode `resize()`

Permet de **changer la taille** d'un **vecteur**, en ajoutant ou supprimant des éléments.

Exemple d'utilisation

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

int main() {
  std::vector<int> vecteur{1, 2, 3, 4, 5};

  vecteur.resize(7); // Ajoute deux éléments à la fin du vecteur

  vecteur.resize(3); // Supprime les derniers éléments du vecteur jusqu'à sa taille 3

  for (auto elt : vecteur) {
    std::cout << elt << " ";
  }

  return 0;
}
```

Output:

```
1 2 3
```

La méthode `resize()` modifie directement le vecteur et n'a pas besoin de retourner un nouveau vecteur.

Syntaxe

```
vecteur.resize(nouvelle_taille);
```

La fonction `resize()` est utilisée pour modifier la taille d'un vecteur en C++. La variable `nouvelle_taille` représente la nouvelle taille souhaitée pour le vecteur. Si la taille est augmentée, de nouveaux éléments seront ajoutés à la fin du vecteur avec des valeurs par défaut; si la taille est réduite, les derniers éléments seront supprimés.

Les valeurs par defaut

1. Pour un vecteur de types numériques tels que `int`, `double`, `float`, etc., les nouveaux éléments auront la valeur `0`.
2. Pour un vecteur de chaînes de caractères (`std::string`), les nouveaux éléments auront la valeur d'une chaîne de caractères vide, c'est-à-dire `""`.
3. Pour un vecteur de types personnalisés (objets d'une classe que vous avez créée), les nouveaux éléments seront des instances de cette classe, créées en utilisant son constructeur par défaut, si elle en a un.

Il est important de noter que vous pouvez également spécifier une valeur par défaut explicite en utilisant une surcharge de la fonction `resize()`. Par exemple :

```
std::vector<int> monVecteur;
monVecteur.resize(5, 42); // Les 5 nouveaux éléments auront la valeur 42
```

Exemples d'utilisation

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

int main() {
```

```
    std::vector<int> v = {1, 2, 3};
    v.resize(6); // {1, 2, 3, 0, 0, 0}
    v.resize(2); // {1, 2}
}
```

Le code démontre comment utiliser la fonction **resize** pour modifier la taille d'un **std::vector** en ajoutant ou enlevant des éléments.

Méthode `reserve()`

Permet de **réserver** de la mémoire pour un certain nombre d'éléments, ce qui améliore les **performances** lors de l'ajout.

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

int main() {
  std::vector<int> myVector;
  myVector.reserve(10);

  for (int i = 0; i < 10; ++i) {
    myVector.push_back(i);
  }

  for (int i = 0; i < 10; ++i) {
    std::cout << myVector[i] << "\\n";
  }

  return 0;
}
```

Syntaxe

```
vecteur.reserve(taille_reservee);
```

Vecteur : un objet de type vector (ex: vector<int>)

taille_reservee : nombre d'éléments à réserver dans le vecteur

La fonction reserve permet d'allouer de l'espace mémoire pour un certain nombre d'éléments afin d'éviter des allocations multiples lors de l'ajout d'éléments.

Exemples d'utilisation

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

int main() {
    std::vector<int> v;
    v.reserve(10); // Réserve de la mémoire pour 10 éléments
}
```

La fonction `reserve()` permet de préallouer de la mémoire pour le conteneur `std::vector`. Elle peut améliorer la performance en évitant les allocations mémoire inutiles lorsqu'on ajoute des éléments.

Utilité

Contrairement à la déclaration d'un vecteur de taille fixe, l'utilisation de `reserve()` ne fixe pas la taille du vecteur, elle réserve simplement un espace mémoire pour le vecteur, mais la taille actuelle du vecteur reste à zéro. Vous pouvez ensuite ajouter des éléments au vecteur, et il grandira au besoin sans devoir réallouer de la mémoire à chaque ajout.

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

int main() {
    std::vector<int> monVecteur;

    monVecteur.reserve(100); // Réserve de la mémoire pour 100 éléments, mais la taill
e actuelle est toujours 0

    for (int i = 0; i < 100; ++i) {
        monVecteur.push_back(i); // Ajoute 100 éléments au vecteur
    }

    std::cout << "Taille du vecteur : " << monVecteur.size() << std::endl; // Affiche
 100

    return 0;
}
```

Contrairement à la déclaration d'un vecteur de taille fixe, l'utilisation de `reserve()` permet au vecteur de s'agrandir dynamiquement au besoin tout en évitant les allocations répétées de mémoire. Cela peut être très utile lorsque vous n'êtes pas sûr à l'avance de la taille exacte du vecteur dont vous aurez besoin.

Méthode `shrink_to_fit()`

Réduit la **capacité** du vecteur pour correspondre à sa **taille actuelle**, libérant ainsi de la **mémoire**.

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

int main() {
  std::vector<int> v = {1, 2, 3, 4, 5};
  v.reserve(20);
  v.shrink_to_fit();

  std::cout << "La capacité du vecteur après avoir utilisé shrink_to_fit() est : " <<
 v.capacity() << std::endl;

  return 0;
}
```

`shrink_to_fit()` est particulièrement utile lorsque l'on souhaite libérer l'espace mémoire non utilisé par un vecteur après plusieurs opérations d'ajout ou de suppression d'éléments.

Syntaxe

```
vecteur.shrink_to_fit();
```

La fonction shrink_to_fit() réduit la capacité du vecteur à sa taille actuelle, libérant ainsi de la mémoire.

Exemples d'utilisation

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

int main() {
    std::vector<int> v = {1, 2, 3};
    v.reserve(10); // Capacité de 10 éléments
    v.shrink_to_fit(); // Réduit la capacité à 3 éléments
}
```

Ici, nous illustrons l'utilisation des **vector** en C ++ avec la méthode **reserve()** pour augmenter la capacité à 10 éléments et **shrink_to_fit()** pour réduire la capacité à 3

éléments.

Ajout et suppression d'éléments

Méthode `push_back()`

La méthode `push_back()` permet d'**ajouter un élément à la fin** d'un **vecteur**.

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

int main() {
  std::vector<int> vecteur;

  // Ajouter des éléments à la fin du vecteur
  vecteur.push_back(1);
  vecteur.push_back(2);
  vecteur.push_back(3);

  // Afficher les éléments du vecteur
  for (int i = 0; i < vecteur.size(); i++) {
    std::cout << vecteur[i] << " ";
  }

  return 0;
}
// Affiche "1 2 3"
```

Syntaxe

```
vector<TYPE> nom_vecteur;
nom_vecteur.push_back(valeur);
```

Méthode	Description
`vector<TYPE>`	Déclaration du vecteur avec un type spécifié
`push_back()`	Ajout d'une valeur à la fin du vecteur

Exemples d'utilisation

```
std::vector<int> entiers;
entiers.push_back(10);
```

```
entiers.push_back(20);
entiers.push_back(30);
```

Les **vecteurs** sont des **conteneurs** de la bibliothèque standard `std` qui peuvent stocker des éléments de même type et dont la taille peut être modifiée dynamiquement. La méthode `push_back()` permet d'ajouter un élément à la fin du vecteur.

Méthode `pop_back()`

La méthode `pop_back()` permet de **supprimer** le **dernier élément** d'un **vecteur**.

Exemple :

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

int main() {
    std::vector<int> mon_vecteur{1, 2, 3, 4, 5};

    // Afficher le contenu du vecteur avant pop_back()
    for (int i : mon_vecteur) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    mon_vecteur.pop_back(); // Supprimer le dernier élément (5)

    // Afficher le contenu du vecteur après pop_back()
    for (int i : mon_vecteur) {
        std::cout << i << " ";
    }

    return 0;
}
```

La méthode `pop_back()` ne retourne pas la valeur supprimée. Elle modifie directement le vecteur.

Syntaxe

```
vector<TYPE> nom_vecteur;
nom_vecteur.pop_back();
```

Exemples d'utilisation

```
std::vector<int> entiers = {10, 20, 30};
entiers.pop_back(); // supprime 30
```

`std::vector` est un exemple de **conteneur** en C++ permettant de conserver des éléments de même type, ici des entiers. La méthode `pop_back()` permet de **supprimer** le dernier élément du vecteur.

Méthode `insert()`

La méthode `insert()` permet d'**insérer** un élément à une position spécifique dans un **vecteur**.

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

int main() {
  std::vector<int> v = {1, 2, 4, 5};

  // insère le nombre 3 à la 3e position
  v.insert(v.begin() + 2, 3);
  for (int x : v) {
    std::cout << x << " ";
  }
  // sortie : 1 2 3 4 5
}
```

Syntaxe

```
vector<TYPE> nom_vecteur;
nom_vecteur.insert(itérateur, élément);
```

Utilisation d'un vecteur en C++ :

- Création d'un vecteur : `vector<TYPE> nom_vecteur;`
- Ajout d'un élément à un vecteur : `nom_vecteur.insert(itérateur, élément);`

Exemples d'utilisation

```
std::vector<int> entiers = {10, 30};
entiers.insert(entiers.begin() + 1, 20); // Insère 20 à la position 1
```

La méthode `insert()` permet d'insérer un élément à une position spécifique dans un vecteur. `begin()` retourne un itérateur pointant vers le premier élément du vecteur et `+1` déplace l'itérateur à la position souhaitée.

Méthode `erase()`

La méthode `erase()` permet de supprimer un **élément** à une **position spécifique** dans un vecteur.

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

int main() {
  std::vector<int> entiers{1, 2, 3, 4, 5};
  entiers.erase(entiers.begin() + 2);

  for (int val : entiers) {
    std::cout << val << " ";
  }
  // Résultat : 1 2 4 5
}
```

Utiliser la méthode `erase()` avec prudence pour éviter les effets indésirables liés à la suppression involontaire d'un élément dans le vecteur.

Syntaxe

```
vector<TYPE> nom_vecteur;
nom_vecteur.erase(itérateur);
```

Exemples d'utilisation

```
std::vector<int> entiers = {10, 20, 30};
entiers.erase(entiers.begin() + 1); // Supprime l'élément à la position 1 (le 20)
```

Les opérations courantes sur les **std::vector** incluent l'ajout, la suppression et la recherche d'éléments. Vous pouvez également utiliser des méthodes telles que

size() et empty() pour obtenir des informations sur le vector.

Parcours des éléments d'un vecteur

Boucle for classique

La boucle **for classique** permet de parcourir les éléments d'un **vecteur** en utilisant un **index**. Cette méthode est similaire au parcours des éléments d'un tableau.

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

int main() {
  std::vector<int> mon_vecteur = {1, 2, 3, 4, 5};

  for (std::size_t i = 0; i < mon_vecteur.size(); ++i) {
    std::cout << mon_vecteur[i] << std::endl;
  }

  return 0;
}
```

Syntaxe

```
for (int i = 0; i < v.size(); i++) {
    // Accéder à l'élément v[i]
}
```

Cette boucle parcourt un vecteur v en utilisant un indice i et permet d'accéder à chaque élément v[i].

Boucle for basée sur une plage

La boucle for basée sur une plage parcourt les éléments d'un **vecteur** sans avoir besoin de connaître l'**index** de chaque élément.

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

int main() {
```

```
    std::vector<int> mon_vecteur = {1, 2, 3, 4, 5};

    for (int element : mon_vecteur) {
        std::cout << element << std::endl;
    }

    return 0;
}
```

Utilisation d'itérateurs

Les **itérateurs** sont des objets qui permettent de naviguer entre les éléments d'un **vecteur**.

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

int main() {
  std::vector<int> v = {1, 2, 3, 4, 5};

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

  return 0;
}
```

Opérations sur les vecteurs

Opérations basiques

Les **opérations basiques** sur les vecteurs incluent l'**affectation** et la **comparaison**.

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

int main() {
  std::vector<int> vec1 = {1, 2, 3};
  std::vector<int> vec2 = {4, 5, 6};

  // Affectation
  vec1 = vec2;

  // Comparaison
  if (vec1 == vec2) {
    std::cout << "Les vecteurs sont égaux" << std::endl;
```

```
  } else {
    std::cout << "Les vecteurs ne sont pas égaux" << std::endl;
  }

  return 0;
}
```

Il est important de rappeler que les vecteurs sont manipulés comme des objets, et non des pointeurs.

Affectation

On peut affecter un **vecteur** à un autre **vecteur** en utilisant l'opérateur `=`.

```
std::vector<int> vect1 = {1, 2, 3};
std::vector<int> vect2 = vect1; // vect2 contient maintenant {1, 2, 3}
```

Les deux vecteurs sont maintenant égaux en contenu, mais ce sont des objets distincts en mémoire.

Comparaison

On peut comparer deux **vecteurs** en utilisant les opérateurs `==`, `!=`, `<`, `>`, `<=`, ou `>=`.

```
std::vector<int> vect1 = {1, 2, 3};
std::vector<int> vect2 = {1, 2, 3};
std::vector<int> vect3 = {2, 3, 4};

bool isEqual = vect1 == vect2; // true
bool isDifferent = vect1 != vect3; // true
bool isLessThan = vect1 < vect3; // true
```

La comparaison de vecteurs se fait en comparant les éléments de chacun. Si tous les éléments sont égaux, les vecteurs sont égaux. Les autres opérateurs se basent sur l'ordre lexicographique des éléments.

Opérations avancées

Concaténation

Pour **concaténer** deux **vecteurs**, on peut utiliser la méthode `insert()`.

```
std::vector<int> vect1 = {1, 2, 3};
std::vector<int> vect2 = {4, 5, 6};

vect1.insert(vect1.end(), vect2.begin(), vect2.end()); // vect1 contient maintenant
 {1, 2, 3, 4, 5, 6}
```

La méthode `insert()` prend trois arguments : la position d'insertion, un itérateur de début et un itérateur de fin.

Trie des éléments

Pour trier les éléments d'un **vecteur**, on peut utiliser la fonction `**std::sort()**`.

```
#include <algorithm> // pour std::sort()

std::vector<int> vect = {3, 1, 2};

std::sort(vect.begin(), vect.end()); // vect contient maintenant {1, 2, 3}
```

Recherche d'éléments

Pour rechercher un élément dans un **vecteur**, on peut utiliser la fonction `std::find()`.

```
#include <algorithm> // pour std::find()

std::vector<int> vect = {1, 2, 3};
int element = 2;

auto it = std::find(vect.begin(), vect.end(), element); // it pointe sur l'élément 2
```

Ici, `auto` est utilisé pour déduire le type de l'itérateur. Si l'élément n'est pas trouvé, `it` sera égal à `vect.end()`. Vous pouvez comparer `it` et `vect.end()` pour savoir si l'élément a été trouvé.

Fonctions utiles de la bibliothèque vector

Fonction `std::swap()`

La fonction `std::swap()` permet d'échanger le contenu de deux **vecteurs**.

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

int main() {
  std::vector<int> vec1 = {1, 2, 3};
  std::vector<int> vec2 = {4, 5, 6};

  std::swap(vec1, vec2);

  for (int i : vec1) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  for (int i : vec2) {
    std::cout << i << " ";
  }
}
```

Pour utiliser la fonction std::swap(), il faut inclure la bibliothèque `<algorithm>`.

Syntaxe

```
std::swap(vector1, vector2);
```

Exemples d'utilisation

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

int main() {
    std::vector<int> vec1 = {1, 2, 3};
    std::vector<int> vec2 = {4, 5, 6};

    std::swap(vec1, vec2);

    // Affiche maintenant 4 5 6
    for (int val : vec1) {
        std::cout << val << " ";
    }
}
```

Cet exemple montre comment utiliser la fonction `std::swap()` pour échanger les éléments de deux vecteurs.

Fonction `std::size()`

La fonction `std::size()` retourne la **taille** d'un **vecteur**.

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

int main() {
    std::vector<int> vecteur = { 1, 2, 3, 4, 5 };

    std::cout << "La taille du vecteur est : " << std::size(vecteur) << std::endl;

    return 0;
}
```

Cette fonction est également disponible pour les tableaux statiques, et fait partie de la bibliothèque `<iterator>`.

Exemples d'utilisation

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

int main() {
    std::vector<int> vec = {1, 2, 3};

    // Affiche la taille du vecteur : 3
    std::cout << "Taille du vecteur : " << std::size(vec) << std::endl;
}
```

Fonction `std::empty()`

La fonction `std::empty()` permet de vérifier si un **vecteur** est vide.

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

int main() {
  std::vector<int> mon_vecteur;

  if (mon_vecteur.empty()) {
    std::cout << "Le vecteur est vide" << std::endl;
  } else {
    std::cout << "Le vecteur n'est pas vide" << std::endl;
  }
```

```
  return 0;
}
```

Syntaxe

```
vector.empty()
```

Cette méthode retourne un booléen pour indiquer si le vecteur est vide (true) ou non (false).

Exemples d'utilisation

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

int main() {
    std::vector<int> vec1 = {1, 2, 3};
    std::vector<int> vec2;

    std::cout << "Le vecteur vec1 est vide ? " << vec1.empty() << std::endl; // Affich
e 0 (false)
    std::cout << "Le vecteur vec2 est vide ? " << vec2.empty() << std::endl; // Affich
e 1 (true)
}
```