LES TABLEAUX

INTRODUCTION AUX TABLEAUX

DÉFINITION

Un **tableau** est une structure de données qui permet de stocker une collection d'éléments de **même type**, contigus en mémoire. Les éléments d'un tableau sont référencés par leur **indice**.

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

int main() {
  int tab[5] = {1, 2, 3, 4, 5};
  cout << tab[0] << endl; // Accès au premier élément du tableau
  return 0;
}
```

UTILISATION ET EXEMPLES

Les **tableaux** sont largement utilisés en programmation pour stocker et manipuler des données. Par exemple, pour stocker les notes d'un étudiant ou les températures d'une semaine.

```
int notes[] = {90, 85, 89, 78};
float temperatures[] = {23.4, 24.8, 22.5, 21.9};
```

Type	Exemple
Entiers	notes d'étudiant
Flottants	températures

Exercice - Tableaux

Les tableaux peuvent également être utilisés pour stocker des chaînes de caractères et d'autres types de données. Les tableaux sont une structure de données linéaire et que chaque élément peut être accédé par un index.

TYPES DE TABLEAUX

TABLEAUX À UNE DIMENSION

Les **tableaux à une dimension** sont des structures de données qui stockent une séquence d'éléments du **même type** les uns à côté des autres en mémoire.

Exemple de déclaration et d'initialisation de tableau :

```
int tableau[5] = {1, 2, 3, 4, 5}; // Tableau de 5 entiers
```

Accès à un élément du tableau :

```
int premier_element = tableau[0]; // Accès au premier élément du tableau (indice 0)
```

DÉCLARATION

Pour déclarer un **tableau à une dimension** en C++, utilisez la syntaxe suivante :

```
type nom_tableau[taille];
```

Composant	Description
type	Le type de données que peut contenir le tableau (int, float, char, etc.)
nom_tableau	Le nom du tableau
taille	Le nombre d'éléments que peut contenir le tableau

N'oubliez pas de préciser la taille du tableau lors de la déclaration. Celle-ci ne peut pas être modifiée par la suite.

INITIALISATION

Pour initialiser un tableau, vous pouvez spécifier les **valeurs** lors de la déclaration :

```
int notes[3] = {85, 90, 78};
```

ACCÈS AUX ÉLÉMENTS

Pour accéder aux éléments d'un tableau, utilisez l'opérateur `[]` et l'**indice** correspondant :

```
int valeur = notes[0];
```

Les indices commencent à 0 et vont jusqu'à la taille du tableau moins 1.

TABLEAUX MULTIDIMENSIONNELS

Les **tableaux multidimensionnels** sont des tableaux dont les éléments sont eux-mêmes des tableaux.

Exemple de déclaration et d'initialisation :

```cpp
#include <iostream>

int main() {
  int tableau[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
  };

  // Afficher les éléments du tableau
  for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 3; ++j) {
      std::cout << tableau[i][j] << " ";
    }
    std::cout << std::endl;
  }
```

DÉCLARATION

Pour déclarer un **tableau multidimensionnel** en C++, utilisez la syntaxe suivante :

```
type nom_tableau[taille1][taille2];
```

INITIALISATION

Pour initialiser un **tableau multidimensionnel**, vous pouvez spécifier les valeurs lors de la déclaration :

```
int grille[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Pour les tableaux multidimensionnels, il est important de définir la taille des dimensions lors de la déclaration.

ACCÈS AUX ÉLÉMENTS

Pour accéder aux éléments d'un **tableau multidimensionnel**, utilisez l'opérateur `[]` et les **indices** correspondants :

```
int valeur = grille[1][2];
```

Ici, grille[1][2] accède au deuxième élément de la deuxième ligne.

OPÉRATIONS SUR LES TABLEAUX

PARCOURS DE TABLEAU

Le parcours de tableau consiste à accéder à chaque **élément** d'un tableau pour effectuer une **opération** ou une **manipulation**.

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

int main() {
    int tableau[] = {1, 2, 3, 4, 5};
    int taille = sizeof(tableau) / sizeof(tableau[0]);

    for(int i = 0; i < taille; i++) {
        cout << "Element #" << i << ": " << tableau[i] << endl;
    }

    return 0;
}
```

Dans cet exemple, on utilise la boucle **for** pour parcourir le tableau. On peut également utiliser une boucle **while** ou **do-while** pour réaliser le parcours d'un tableau.

BOUCLE FOR

Une boucle `for` est la méthode la plus courante pour parcourir un **tableau** en C++.

```
for (int i = 0; i < taille; i++) {
    cout << tableau[i] << endl;
}
```

BOUCLE WHILE

Une boucle `while` peut aussi être utilisée pour parcourir un **tableau**.

```
int i = 0;
while (i < taille) {
    cout << tableau[i] << endl;
    i++;
}
```

BOUCLE DO/WHILE

Un parcours de tableau avec une boucle `do/while` est aussi possible.

```
int i = 0;
do {
    cout << tableau[i] << endl;
    i++;
} while (i < taille);
```

La boucle do/while est souvent utilisée lorsqu'on souhaite s'assurer qu'une instruction soit exécutée au moins une fois avant de vérifier la condition.

RECHERCHE D'ÉLÉMENTS

RECHERCHE LINÉAIRE

La **recherche linéaire** consiste à parcourir un tableau pour trouver un élément spécifique.

```
int indice = -1;
for (int i = 0; i < taille; i++) {
    if (tableau[i] == valeur_recherchee) {
        indice = i;
        break;
    }
}
```

La recherche linéaire est simple et facile à implémenter, mais elle peut être inefficace pour les grands tableaux.

RECHERCHE BINAIRE (TABLEAU TRIÉ)

La **recherche binaire**, plus rapide que la linéaire, s'applique sur un **tableau trié**.

```c
int recherche_binaire(int tableau[], int taille, int valeur_recherchee) {
    int gauche = 0;
    int droite = taille - 1;
    while (gauche <= droite) {
        int milieu = (gauche + droite) / 2;
        if (tableau[milieu] == valeur_recherchee) {
            return milieu;
        } else if (tableau[milieu] < valeur_recherchee) {
            gauche = milieu + 1;
        } else {
            droite = milieu - 1;
        }
    }
    return -1;
}
```

L'idée derrière la recherche binaire est de diviser l'intervalle de recherche par deux à chaque étape en comparant l'élément central à la valeur recherchée.

TRI DE TABLEAUX

TRI À BULLES

Le tri à bulles consiste à comparer les éléments **adjacents** d'un tableau et les échanger si nécessaire.

```
for (int i = 0; i < taille - 1; i++) {
    for (int j = 0; j < taille - 1 - i; j++) {
        if (tableau[j] > tableau[j + 1]) {
            swap(tableau[j], tableau[j + 1]);
        }
    }
}
```

Le tri à bulles compare chaque paire d'éléments et met en évidence l'importance de la fonction swap.

TRI PAR SÉLECTION

Le **tri par sélection** fonctionne en sélectionnant le **plus petit élément** à chaque itération et en l'échangeant avec l'élément à la **position courante**.

```
for (int i = 0; i < taille - 1; i++) {
    int indice_min = i;
    for (int j = i + 1; j < taille; j++) {
        if (tableau[j] < tableau[indice_min]) {
            indice_min = j;
        }
    }
    swap(tableau[i], tableau[indice_min]);
}
```

Le tri par sélection est l'une des méthodes de tri les plus simples à comprendre et à implémenter, bien qu'elle ne soit pas la plus rapide pour les grands ensembles de données.

TRI PAR INSERTION

Le tri par insertion fonctionne en **insérant** un élément à sa place dans la **partie triée** du tableau.

```
for (int i = 1; i < taille; i++) {
    int valeur_temp = tableau[i];
    int j = i - 1;
    while (j >= 0 && tableau[j] > valeur_temp) {
        tableau[j + 1] = tableau[j];
        j--;
    }
    tableau[j + 1] = valeur_temp;
}
```

La partie non triée du tableau est parcourue, et pour chaque élément, on trouve sa place dans la partie triée en déplaçant les autres éléments pour faire de la place.

PASSAGE DE TABLEAUX AUX FONCTIONS

SYNTAXE ET EXEMPLES

Pour passer un tableau à une fonction, utilisez le nom du tableau sans indices. La fonction doit connaître la taille du tableau.

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

void printArray(int myArray[], int size);

int main() {
  int arr[] = {1, 2, 3, 4, 5};
  int n = sizeof(arr)/sizeof(arr[0]);
  printArray(arr, n);
}

void printArray(int myArray[], int size) {
  for (int i = 0; i < size; i++) {
    cout << myArray[i] << " ";
  }
```

PASSAGE PAR RÉFÉRENCE

Le passage par référence permet de modifier les éléments du tableau d'origine.

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

void doubleArray(int (&arr)[5]);

int main() {
  int arr[] = {1, 2, 3, 4, 5};
  doubleArray(arr);
}

void doubleArray(int (&arr)[5]) {
  for (int i = 0; i < 5; i++) {
    arr[i] *= 2;
  }
}
```

PASSAGE PAR POINTEUR

Le passage par **pointeur** permet de manipuler les adresses mémoire du tableau.

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

void printPointerArray(int *arr, int size);

int main() {
  int arr[] = {1, 2, 3, 4, 5};
  int n = sizeof(arr)/sizeof(arr[0]);
  printPointerArray(arr, n);
}

void printPointerArray(int *arr, int size) {
  for (int i = 0; i < size; i++) {
    cout << *(arr + i) << " ";
  }
```

TABLEAUX DYNAMIQUES

UTILISATION DE LA MÉMOIRE DYNAMIQUE

Les **tableaux dynamiques** utilisent la mémoire dynamique, qui permet de **modifier la taille des tableaux** pendant l'exécution du programme.

Avantages	Inconvénients
Allocation de mémoire ajustée	Complexité de gestion supplémentaire
Utilisation efficace de l'espace mémoire	Risques de fragmentation de la mémoire

```
int *tableauDynamique = new int[10]; // Allocation de mémoire pour 10 entiers
delete[] tableauDynamique; // Libération de la mémoire allouée
```

CRÉATION DE TABLEAUX DYNAMIQUES

```
int* tableau = new int[10]; // Crée un tableau dynamique de 10 entiers
delete[] tableau; // Libère la mémoire allouée au tableau
```

Points clés:

- Utilisation de `new` pour allouer de la mémoire
- Utilisation de `delete[]` pour libérer la mémoire

Il faut toujours libérer la mémoire allouée pour éviter les fuites de mémoire.

SUPPRESSION DE TABLEAUX DYNAMIQUES

```
delete[] tableau; // Libère la mémoire allouée au tableau
tableau = nullptr; // Met à jour le pointeur pour éviter de pointer vers une zone mémoire libérée
```

Il est important de toujours libérer la mémoire allouée manuellement après utilisation pour éviter les fuites de mémoire dans le programme.

UTILISATION AVANCÉE DES TABLEAUX

TABLEAUX DE CHAÎNES DE CARACTÈRES

Les **tableaux** de chaînes de caractères sont des **tableaux de tableaux** de caractères.

```
char tableau[3][10] = {"Hello", "World", "C++"};
```

TABLEAUX DE STRUCTURES

Les tableaux de structures sont des tableaux dont chaque élément est une **structure**.

```
struct Point {
  int x;
  int y;
};

Point tableau[5];
```

Chaque élément du tableau peut être accédé et modifié comme une structure simple en utilisant l'opérateur point `.` avec l'indice de l'élément.

TABLEAUX DE STRUCTURES : INITIALISATION

Il est possible d'**initialiser** un tableau de structures en déclarant **explicitement** les valeurs.

```
struct Point {
  int x;
  int y;
};

Point tableau[2] = {{1, 2}, {3, 4}};
```

Pour les tableaux de structure, chaque élément du tableau est une instance de la structure, et les valeurs doivent être sous forme de paires d'accolades `{}` pour chaque élément.

TABLEAUX ET POINTEURS

Les pointeurs peuvent être utilisés pour accéder aux éléments d'un **tableau**.

```
int tableau[3] = {1, 2, 3};
int *ptr = tableau;
```

`tableau` est l'adresse mémoire de la première case du tableau.

ACCÈS AUX ÉLÉMENTS AVEC POINTEURS

L'accès aux éléments d'un tableau avec des **pointeurs** est similaire à l'utilisation de l'opérateur d'**indexation** `[]`.

```
int val1 = *ptr;       // Val1 = 1
int val2 = *(ptr + 1); // Val2 = 2
int val3 = ptr[2];     // Val3 = 3
```

Les pointeurs sont des variables qui contiennent l'adresse mémoire d'une autre variable et peuvent être utilisés pour accéder à ces valeurs.

POINTEURS ET TABLEAU MULTIDIMENSIONNEL

Pour pointer sur un tableau multidimensionnel, il faut utiliser des **pointeurs de pointeurs**.

```
int tableau[2][3] = {{1, 2, 3}, {4, 5, 6}};
int (*ptr)[3] = tableau; // Pointeur sur tableau de 3 éléments
```

ACCÈS AUX ÉLÉMENTS AVEC POINTEURS MULTIDIMENSIONNELS

On peut accéder aux éléments d'un tableau **multidimensionnel** en utilisant des **pointeurs**.

```
int val1 = *(*(ptr) + 1);     // val1 = 2
int val2 = *(*(ptr + 1) + 2); // val2 = 6
```

Pour accéder aux éléments d'un tableau multidimensionnel avec des pointeurs, on utilise la déréférence deux fois.