Chapitre 5 : Les Pointeurs et les Tableaux

Objectifs pédagogiques :

- Comprendre la notion d'adresse mémoire et de pointeur
- Maîtriser la différence entre pointeurs et références
- Savoir manipuler des tableaux statiques et dynamiques
- Créer et utiliser des tableaux multidimensionnels pour le calcul scientifique
- Éviter les erreurs classiques de gestion mémoire

1. Introduction à la mémoire

Avant d'aborder les pointeurs, il est essentiel de comprendre comment les variables sont stockées en mémoire.

1.1. La mémoire vive (RAM)

La mémoire peut être visualisée comme une immense succession de "cases" numérotées, chacune pouvant contenir un octet (8 bits).

Chaque variable que nous déclarons occupe un certain nombre de ces cases. Par exemple :

- Un char occupe 1 case (1 octet)
- Un int occupe généralement 4 cases (4 octets)
- Un double occupe généralement 8 cases (8 octets)

1.2. L'opérateur & (adresse de)

L'opérateur unaire & permet d'obtenir l'adresse mémoire d'une variable.

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

int main() {
    int age = 25;
    double temperature = 36.6;

    cout << "Valeur de age : " << age << endl;
    cout << "Adresse de age : " << &age << endl;  // Affiche un hexadécimal (ex: 0x7ffd5e3a4)

    cout << "Valeur de temperature : " << temperature << endl;
    cout << "Adresse de temperature : " << &temperature << endl;

    return 0;
}
```

Important : L'adresse affichée est en hexadécimal (base 16). Chaque exécution peut donner des adresses différentes.

2. Les pointeurs

2.1. Définition

Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable.

2.2. Déclaration et initialisation

Syntaxe :

```
type* nom_pointeur;
```

Exemple :

```
int* p;           // Pointeur vers un entier (non initialisé)
double* ptr;      // Pointeur vers un double

// Initialisation avec l'adresse d'une variable
int age = 25;
int* p_age = &age;  // p_age contient l'adresse de age

cout << "age = " << age << endl;
cout << "Adresse de age = " << &age << endl;
cout << "p_age = " << p_age << endl;  // Même adresse que &age
```

2.3. L'opérateur * (déréférencement)

L'opérateur * permet d'accéder à la valeur pointée par un pointeur.

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

int main() {
    int age = 25;
    int* p_age = &age;  // p_age pointe vers age

    cout << "age = " << age << endl;           // 25
    cout << "p_age = " << p_age << endl;       // adresse de age
    cout << "*p_age = " << *p_age << endl;     // 25 (la valeur pointée)

    // Modification via le pointeur
    *p_age = 30;  // Modifie la valeur de age via le pointeur

    cout << "Après modification :" << endl;
    cout << "age = " << age << endl;           // 30
```

```
    cout << "*p_age = " << *p_age << endl;      // 30

    return 0;
}
```

2.4. Pointeur nul

Un pointeur peut ne pointer vers rien (valeur nulle).

```
int* p = nullptr;  // C++11 : pointeur nul (recommandé)
// ou
int* q = NULL;     // ancienne façon (C-style)
// ou
int* r = 0;        // déconseillé

// Toujours vérifier avant d'utiliser
if (p != nullptr) {
    *p = 10;  // Sûr
}
```

3. Les références

3.1. Définition

Une référence est un alias (un autre nom) pour une variable existante. Contrairement aux pointeurs, une référence ne peut pas être réaffectée à une autre variable.

Syntaxe :

```
type& nom_reference = variable;
```

```
// Syntaxe : type& nom_reference = variable;
int age = 25;
int& ref_age = age;  // ref_age est une référence vers age

ref_age = 30;  // Modifie age (qui vaut maintenant 30)

cout << "age = " << age << endl;        // 30
cout << "ref_age = " << ref_age << endl; // 30
```

3.2. Différences pointeurs vs références

Caractéristique	Pointeur	Référence
Peut être nul	Oui	Non (doit être initialisée)
Peut être réaffecté	Oui	Non

Caractéristique	Pointeur	Référence
Syntaxe d'accès	*p pour déréférencer	Directe (comme variable)
Calcul arithmétique	Oui	Non

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

int main() {
    int a = 10;
    int b = 20;

    // Pointeur
    int* p = &a;
    p = &b;  // OK : on peut changer ce que pointe p
    *p = 30; // Modifie b

    // Référence
    int& r = a;
    // r = &b;  // ERREUR : on ne peut pas réaffecter une référence
    r = 40;    // Modifie a (r est juste un autre nom pour a)

    cout << "a = " << a << ", b = " << b << endl;

    return 0;
}
```

3.3. Utilisation typique des références

Les références sont très utilisées comme paramètres de fonctions pour éviter les copies.

4. Tableaux statiques

4.1. Déclaration et initialisation

Un tableau est une collection d'éléments de même type, stockés consécutivement en mémoire.

Syntaxe :

```
Type nom[taille]
```

Exemple :

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

int main() {
    // Déclaration d'un tableau de 5 entiers
    int mesures[5];

    // Initialisation à la déclaration
    double temperatures[4] = {18.5, 22.3, 19.8, 21.0};

    // Initialisation partielle (les autres sont à 0)
    int compteurs[10] = {0};  // Tous à 0

    // Initialisation sans taille (déduite)
    char codes[] = {'A', 'B', 'C', 'D'};  // taille = 4

    // Accès aux éléments (index de 0 à taille-1)
    cout << "Première température : " << temperatures[0] << endl;
    cout << "Dernière température : " << temperatures[3] << endl;

    // Modification
    temperatures[1] = 23.5;

    return 0;
}
```

4.2. Parcours de tableaux

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

int main() {
    const int NB_CAPTEURS = 5;
    double valeurs[NB_CAPTEURS];

    // Saisie des valeurs
    cout << "Entrez " << NB_CAPTEURS << " mesures :" << endl;
    for (int i = 0; i < NB_CAPTEURS; i++) {
        cout << "Capteur " << i << " : ";
        cin >> valeurs[i];
    }
```

```
    // Affichage
    cout << "\nMesures enregistrées :" << endl;
    for (int i = 0; i < NB_CAPTEURS; i++) {
        cout << "v[" << i << "] = " << valeurs[i] << endl;
    }

    // Calcul de la moyenne
    double somme = 0.0;
    for (int i = 0; i < NB_CAPTEURS; i++) {
        somme += valeurs[i];
    }
    double moyenne = somme / NB_CAPTEURS;
    cout << "Moyenne = " << moyenne << endl;

    return 0;
}
```

4.3. Initialisation avec C++11

```
// C++11 permet l'initialisation uniforme
double mesures[] {12.5, 13.8, 11.2, 14.1};  // Pas besoin de =

// Pour les tableaux de taille fixe
const int TAILLE = 100;
double buffer[TAILLE] {};  // Tous les éléments à 0
```

4.4. Problème classique : dépassement d'indice

```
int tableau[5] = {10, 20, 30, 40, 50};

// ERREUR : indice 5 n'existe pas (valides : 0 à 4)
tableau[5] = 60;  // Comportement indéfini !

// Le compilateur ne détecte pas cette erreur !
// Cela peut écraser d'autres variables ou planter le programme.
```

Règle : Vérifiez toujours que vos indices sont dans les limites du tableau.

4.5. Relation entre tableaux et pointeurs

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

int main() {
    int mesures[5] = {10, 20, 30, 40, 50};

    cout << "mesures = " << mesures << endl;      // Adresse du premier élément
```

```
    cout << "&mesures[0] = " << &mesures[0] << endl; // Même adresse
    cout << "*mesures = " << *mesures << endl;    // 10 (valeur du premier)

    // Accès via arithmétique de pointeurs
    cout << "*(mesures + 2) = " << *(mesures + 2) << endl;  // 30 (identique à mesures[2])

    return 0;
}
```

Explication : mesures[i] est équivalent à *(mesures + i)

4.6. Parcours d'un tableau avec pointeur

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

int main() {
    const int TAILLE = 5;
    double mesures[TAILLE] = {10.5, 20.3, 15.8, 12.1, 18.9};

    // Méthode 1 : avec indices
    cout << "Avec indices :" << endl;
    for (int i = 0; i < TAILLE; i++) {
        cout << mesures[i] << " ";
    }
    cout << endl;

    // Méthode 2 : avec pointeur
    cout << "Avec pointeur :" << endl;
    for (double* p = mesures; p < mesures + TAILLE; p++) {
        cout << *p << " ";
    }
    cout << endl;

    // Méthode 3 : range-based for (C++11)
    cout << "Range-based for :" << endl;
    for (double val : mesures) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

5. Tableaux dynamiques

5.1. Allocation dynamique avec new et delete

Les tableaux statiques ont une taille fixe déterminée à la compilation. Parfois, on ne connaît la taille qu'à l'exécution.

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

int main() {
    int taille;

    cout << "Combien de mesures voulez-vous enregistrer ? ";
    cin >> taille;

    // Allocation dynamique
    double* mesures = new double[taille];

    // Utilisation comme un tableau normal
    for (int i = 0; i < taille; i++) {
        cout << "Mesure " << i << " : ";
        cin >> mesures[i];
    }

    cout << "\nMesures enregistrées :" << endl;
    for (int i = 0; i < taille; i++) {
        cout << "mesures[" << i << "] = " << mesures[i] << endl;
    }

    // Calcul de la moyenne
    double somme = 0.0;
    for (int i = 0; i < taille; i++) {
        somme += mesures[i];
    }
    cout << "Moyenne = " << somme / taille << endl;

    // IMPORTANT : libérer la mémoire !
    delete[] mesures;
    mesures = nullptr;  // Bonne pratique

    return 0;
}
```

Règles d'or de l'allocation dynamique :

- Toujours libérer la mémoire avec delete[] (pour les tableaux)
- Mettre le pointeur à nullptr après libération
- Vérifier que new a réussi (en C++ moderne, il lance une exception en cas d'échec)
- Ne pas mélanger new/delete avec malloc/free

6. Tableaux multidimensionnels

6.1. Tableaux 2D statiques

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

int main() {
    // Déclaration d'une matrice 3x4
    double matrice[3][4];

    // Initialisation
    double identite[3][3] = {
        {1.0, 0.0, 0.0},
        {0.0, 1.0, 0.0},
        {0.0, 0.0, 1.0}
    };

    // Accès : matrice[ligne][colonne]
    cout << "Élément (1,1) : " << identite[1][1] << endl;  // 1.0

    // Remplissage
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            matrice[i][j] = i * 4 + j;
        }
    }

    // Affichage
    cout << "\nMatrice 3x4 :" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            cout << matrice[i][j] << "\t";
        }
        cout << endl;
    }
    return 0;
}
```

6.2. Organisation en mémoire

Les tableaux 2D sont stockés en mémoire ligne par ligne (row-major order).

6.3. Tableaux 2D dynamiques

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

int main() {
    int lignes, colonnes;

    cout << "Entrez le nombre de lignes : ";
    cin >> lignes;
    cout << "Entrez le nombre de colonnes : ";
    cin >> colonnes;

    // Allocation d'un tableau de pointeurs (lignes)
    double** matrice = new double*[lignes];

    // Allocation de chaque ligne
    for (int i = 0; i < lignes; i++) {
        matrice[i] = new double[colonnes];
    }

    // Remplissage
    cout << "\nEntrez les éléments de la matrice :" << endl;
    for (int i = 0; i < lignes; i++) {
        for (int j = 0; j < colonnes; j++) {
            cout << "matrice[" << i << "][" << j << "] = ";
            cin >> matrice[i][j];
        }
    }

    // Affichage
    cout << "\nMatrice saisie :" << endl;
    for (int i = 0; i < lignes; i++) {
        for (int j = 0; j < colonnes; j++) {
            cout << matrice[i][j] << "\t";
        }
        cout << endl;
    }

    // Libération mémoire (dans l'ordre inverse)
    for (int i = 0; i < lignes; i++) {
```

```
        delete[] matrice[i];  // Libère chaque ligne
    }
    delete[] matrice;  // Libère le tableau de pointeurs

    return 0;
}
```

6.4. Visualisation de l'allocation 2D

```
matrice  ->  [ptr_ligne0]  ->  [col0] [col1] [col2] ...
             [ptr_ligne1]  ->  [col0] [col1] [col2] ...
             [ptr_ligne2]  ->  [col0] [col1] [col2] ...
```

7. Synthèse

Concept	Syntaxe	Usage
Adresse d'une variable	`&x`	Obtenir l'adresse
Pointeur	`int* p = &x;`	Stocker une adresse
Déréférencement	`*p`	Accéder à la valeur pointée
Référence	`int& r = x;`	Alias vers une variable
Tableau statique	`double t[10];`	Taille fixe connue à la compilation
Tableau dynamique	`double* t = new double[n];`	Taille déterminée à l'exécution
Tableau 2D statique	`double m[3][4];`	Matrice taille fixe
Tableau 2D dynamique	`double** m = new double*[l];`	Matrice taille variable
Libération mémoire	`delete[] p;`	Obligatoire après `new[]`