Les pointeurs en C++

Rappel sur les passage de variable en C++

1. Introduction aux pointeurs

- Définition d'un pointeur
- Différence entre variables classiques et pointeurs
- Pourquoi utiliser des pointeurs ? (Efficacité, manipulation dynamique de la mémoire, etc.)

2. Déclaration et initialisation des pointeurs

- Syntaxe de base (`int* ptr;`)
- Initialisation (`int a = 10; int* ptr = &a;`)
- L'opérateur d'adressage (`&`) et de déréférencement (`*`)

3. Manipulation des pointeurs

- Modification de la valeur pointée
- Changement d'adresse stockée dans le pointeur
- Pointeurs et types (`int*`, `double*`, `char*`)

4. Pointeurs et tableaux

- Un tableau comme un pointeur (`int arr[5]`)
- Arithmétique des pointeurs (`ptr++`, `ptr--`, `ptr + n`)
- Différence entre `arr` et `&arr[0]`

5. Allocation dynamique de mémoire

- Opérateurs `new` et `delete`
- Allocation et libération de mémoire (`int* ptr = new int; delete ptr;`)
- Allocation de tableaux dynamiques (`new int[n]`, `delete[] ptr;`)

6. Pointeurs et fonctions

- Passage de pointeurs en argument (`void f(int* p)`)
- Retourner un pointeur depuis une fonction
- Différence entre passage par valeur, par référence et par pointeur

7. Pointeurs et structures

- Pointeurs vers des structures (`struct Point* p`)
- Accès aux membres via `->`
- Création dynamique de structures

8. Pointeurs et classes

- Pointeurs vers objets
- Opérateur `this`
- Allocation dynamique d’objets

9. Pointeurs de fonctions

- Définition et utilisation (`void (*funcPtr)(int)`)
- Passer un pointeur de fonction en paramètre

10. Pointeurs intelligents (Smart Pointers)

- Introduction (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`)
- Avantages et différences avec les pointeurs classiques

11. Problèmes courants et bonnes pratiques

- Dangling pointers (pointeurs suspendus)
- Memory leaks (fuites de mémoire)
- Null pointers et vérifications (`if (ptr != nullptr)`)

12. Exercices et applications

- Exercices pratiques sur chaque section
- Implémentation d’une liste chaînée
- Gestion mémoire dans un mini-projet

Passage des variables

1. Passage par valeur

Quand une variable est passée par **valeur**, une copie est créée. Toute modification dans la fonction ne modifie pas l'original.

```
#include <iostream>

void modifierValeur(int x) {
  x = 10; // Modifie la copie, pas la variable d'origine
}

int main() {
  int a = 5;
  modifierValeur(a);
  std::cout << "a = " << a << std::endl; // Toujours 5
  return 0;
}
```

2. Passage par référence

Quand une variable est passée par **référence**, la fonction travaille directement sur l'originale.

```
#include <iostream>

void modifierValeur(int& x) {
  x = 10; // Modifie la variable originale
}

int main() {
  int a = 5;
  modifierValeur(a);
  std::cout << "a = " << a << std::endl; // Maintenant 10
  return 0;
}
```

3. Passage par référence constante

Si on ne veut **pas modifier** la variable passée mais éviter la copie (utile pour les objets), on utilise `const`

```
#include <iostream>


void afficherValeur(const int& x) {
  std::cout << "Valeur : " << x << std::endl;
}
int main() {
  int a = 5;
  afficherValeur(a);
  return 0;
}
```

Définition des pointeurs en C++

Un **pointeur** est une variable spéciale qui **stocke l'adresse mémoire** d'une autre variable. Contrairement aux variables classiques qui stockent directement des valeurs, un pointeur **référence une adresse en mémoire** où se trouve une donnée.

Explication avec une analogie :

Imagine un **pointeur comme une carte contenant l'adresse d'une maison** (la variable). Si tu as cette carte, tu peux aller directement voir ce qu'il y a à cette adresse.

Syntaxe de base :

Un pointeur est déclaré en spécifiant son type suivi de `*` :

```
int a = 10; // Une variable entière

int* ptr = &a; // Un pointeur qui stocke l'adresse de a
```

- `&a` : Obtient l'adresse mémoire de a
- `ptr` : Contient cette adresse
- `*ptr` : Permet d'accéder à la valeur stockée à cette adresse (déréférencement)

Illustration mémoire :

Si a est stocké à l'adresse **0x100**, alors :

Nom	Valeur	Adresse
a ptr	10 0x100	0x100 0x200

Avec `*ptr`, on accède directement à la valeur **10**.

Pourquoi utiliser des pointeurs ?

- Accéder et manipuler directement la mémoire
- Passer des **paramètres par adresse** aux fonctions
- Travailler avec des **tableaux et des chaînes de caractères**
- Gérer la **mémoire dynamique** (`new` et `delete`)
- Implémenter des **structures complexes** comme les listes chaînées

En résumé, un **pointeur est un outil puissant mais aussi risqué**, car une mauvaise gestion peut provoquer des erreurs comme des **fuites mémoire** ou des **pointeurs suspendus**.

Différence entre variables classiques et pointeurs :

Caractéristique	Variables classiques	Pointeurs
Définition	Stockent directement une valeur.	Stockent une **adresse mémoire** d'une autre variable.
Syntaxe de déclaration	`int x = 10;`	`int* ptr = &x;` (pointeur vers un entier)
Stockage	Contient une valeur spécifique (ex: `10`).	Contient une **adresse mémoire** (ex: `0x100`).
Modification	Modifier la variable change sa valeur	Modifier le pointeur change l'adresse qu'il référence.
Opérations associées	Opérations classiques (`+, -, *, /, etc.`).	Peut être **déréférencé** (`*ptr`) et supporte l'**arithmétique des pointeurs** (`ptr++`).
Utilisation	Simplicité et sécurité pour manipuler des données.	Utile pour la **mémoire dynamique**, les **structures de données**, et l'**optimisation de la mémoire**.
Allocation mémoire	Stocké automatiquement sur la **pile**.	Peut être **dynamique** (sur le **tas**) via `new`.

Exemple de fonctionnement	`int x = 10;`
	`cout << x;`
	(Affiche `10`).
		`int x = 10;`
		`int* ptr = &x;`
		`cout << *ptr;`
		(Affiche aussi `10`, mais via un accès indirect).

→ Une **variable classique** contient directement une valeur.

→ Un **pointeur** contient l'adresse d'une autre variable et permet d'y accéder indirectement.

→ Un **pointeur permet de manipuler la mémoire de manière flexible**, mais nécessite une gestion rigoureuse (allocation, libération, vérification des pointeurs nuls).

Exemple 1 : Variable classique vs Pointeur

```cpp
#include <iostream>

int main() {
    int x = 10;    // Déclaration d'une variable classique
    int* ptr = &x;  //Déclaration d'un pointeur qui stocke l'adresse de x

    std::cout << "Valeur de x : " << x << std::endl;    // Affiche 10
    std::cout << "Adresse de x : " << &x << std::endl;      // Affiche l'adresse mémoire de x
    std::cout << "Valeur du pointeur ptr : " << ptr << std::endl; // Contient l'adresse de x
    std::cout << "Valeur pointée par ptr (*ptr) : " << *ptr << std::endl; // Affiche la valeur de x (10)

    // Modification de x via le pointeur
    *ptr = 20;
    std::cout << "\nAprès modification via le pointeur :" << std::endl;
    std::cout << "Valeur de x : " << x << std::endl; // Affiche 20

    return 0;
}
```

Affichage dans le moniteur :

Valeur de x : 10

Adresse de x : 0x61ff08

Valeur du pointeur ptr : 0x61ff08

Valeur pointée par ptr (*ptr) : 10

Après modification via le pointeur :

Valeur de x : 20

Quelques explications :

1. La variable `x` contient directement la valeur `10`.
2. Le pointeur `ptr` stocke **l'adresse mémoire** de `x`.
3. `*ptr` permet **d'accéder et de modifier la valeur de** `x` via le pointeur.
4. En changeant `*ptr = 20;`, on modifie `x` **sans l'utiliser directement**.

Exemple 2 : Différence entre une variable classique et un pointeur dans une fonction

Voici un exemple avec une fonction pour illustrer comment les pointeurs permettent **de modifier une variable en dehors de la fonction**.

```
#include <iostream>
// Fonction qui tente de modifier une variable classique (ça ne fonctionne pas)
void changerValeur(int a) {
  a = 50;
}
// Fonction qui modifie une variable via un pointeur (ça fonctionne)
void changerValeurAvecPointeur(int* a) {
  *a = 50;
}
int main() {
  int x = 10;
  std::cout << "Avant appel de changerValeur, x = " << x << std::endl;
  changerValeur(x);
  std::cout << "Après appel de changerValeur, x = " << x << std::endl; // x reste 10

  std::cout << "\nAvant appel de changerValeurAvecPointeur, x = " << x << std::endl;
  changerValeurAvecPointeur(&x);
  std::cout << "Après appel de changerValeurAvecPointeur, x = " << x << std::endl; // x est modifié à 50

  return 0;
}
```

Affichage dans le moniteur :

Avant appel de changerValeur, x = 10

Après appel de changerValeur, x = 10

Avant appel de changerValeurAvecPointeur, x = 10

Après appel de changerValeurAvecPointeur, x = 50

Quelques explications :

1. Dans `changerValeur(int a)`, la variable a est une **copie** de x, donc la modification ne s'applique pas à x.
2. Dans `changerValeurAvecPointeur(int* a)`, on passe **l'adresse de** x, donc la modification affecte directement la mémoire de x.

Exemple 3 : Un tableau en C++ est en réalité un pointeur vers son premier élément. On peut manipuler un tableau à l'aide d'un pointeur.

```
#include <iostream>


int main() {
  int tab[] = {10, 20, 30, 40, 50};
   int* ptr = tab;  // Un tableau est un pointeur vers son premier élément


   std::cout << "Première valeur (tab[0]) : " << *ptr << std::endl; // 10
     std::cout << "Deuxième valeur (tab[1]) : " << *(ptr + 1) << std::endl; // 20


   // Parcourir le tableau avec un pointeur
     std::cout << "\nParcourir le tableau avec un pointeur :" << std::endl;
   for (int i = 0; i < 5; i++) {
     std::cout << "tab[" << i << "] = " << *(ptr + i) << std::endl;
   }


   return 0;
}
```

Affichage dans le moniteur :

Première valeur (tab[0]) : 10

Deuxième valeur (tab[1]) : 20

Parcourir le tableau avec un pointeur :

tab[0] = 10

tab[1] = 20

tab[2] = 30

tab[3] = 40

tab[4] = 50

Quelques explications :

1. `tab` est un pointeur vers le premier élément du tableau.
2. `*(ptr + i)` permet d'accéder aux éléments du tableau sans utiliser `tab[i]`.
3. L'**arithmétique des pointeurs** (`ptr + i`) permet de se déplacer dans la mémoire.

Exemple 4 : Allocation Dynamique avec `new` et `delete`

Les pointeurs permettent d'allouer de la mémoire **dynamiquement**, c'est-à-dire pendant l'exécution du programme.

Exemple : programme d'allocation et libération d'un entier

```
#include <iostream>

int main() {
  int* ptr = new int; // Allocation dynamique d'un entier

  *ptr = 42; // Affectation d'une valeur
  std::cout << "Valeur de ptr : " << *ptr << std::endl; // Affiche 42

  delete ptr; // Libération de la mémoire
  ptr = nullptr; // Bonne pratique : éviter un pointeur suspendu

  return 0;
}
```

Affichage dans le moniteur :

Valeur de ptr : 42

Quelques explications :

1. `new int` alloue un espace mémoire pour un `int` sur le **tas** (heap).
2. `delete ptr` libère cette mémoire pour éviter une **fuite mémoire**.
3. `ptr = nullptr;` évite les **pointeurs suspendus** (dangling pointers).

Exemple 5 : **Allocation Dynamique d'un Tableau**

On peut aussi allouer un **tableau dynamique**, ce qui est utile lorsque la taille du tableau est inconnue à la compilation.

Exemple : Création dynamique d'un tableau

```
#include <iostream>

int main() {
  int n;
  std::cout << "Entrez la taille du tableau : ";
  std::cin >> n;
  int* tab = new int[n]; // Allocation dynamique d'un tableau

  // Remplissage du tableau
  for (int i = 0; i < n; i++) {
    tab[i] = (i + 1) * 10;
  }

  // Affichage du tableau
  std::cout << "Contenu du tableau dynamique :" << std::endl;
  for (int i = 0; i < n; i++) {
    std::cout << "tab[" << i << "] = " << tab[i] << std::endl;
  }
  delete[] tab; // Libération de la mémoire
  tab = nullptr;

  return 0;
}
```

Affichage sur le moniteur :

Entrez la taille du tableau : 3

Contenu du tableau dynamique :

tab[0] = 10

tab[1] = 20

tab[2] = 30

Quelques explications :

1. `new int[n]` alloue dynamiquement un tableau de n entiers.
2. `delete[] tab` libère la mémoire allouée pour éviter une fuite mémoire.
3. `tab = nullptr;` est une bonne pratique après `delete`.

Exemple 6 : Liste Chaînée avec Pointeurs

Les pointeurs permettent de créer des **structures dynamiques**, comme les **listes chaînées**.

Exemple : Implémentation d'une liste chaînée simple

```
#include <iostream>

// Définition d'un nœud de la liste
struct Node {
  int data;
  Node* next;
};

// Fonction pour ajouter un élément à la liste
void ajouterEnTete(Node*& head, int valeur) {
  Node* newNode = new Node;
  newNode->data = valeur;
  newNode->next = head;
  head = newNode;
}

// Fonction pour afficher la liste
void afficherListe(Node* head) {
  Node* temp = head;
  while (temp != nullptr) {
    std::cout << temp->data << " -> ";
    temp = temp->next;
  }
  std::cout << "NULL" << std::endl;
}

// Libération de la mémoire
void libererListe(Node*& head) {
  while (head != nullptr) {
    Node* temp = head;
    head = head->next;
    delete temp;
  }
}
```

```
int main() {
  Node* liste = nullptr; // Déclaration d'une liste vide

  ajouterEnTete(liste, 10);
  ajouterEnTete(liste, 20);
  ajouterEnTete(liste, 30);

  std::cout << "Liste chaînée : ";
  afficherListe(liste);

  libererListe(liste); // Nettoyage mémoire

  return 0;
}
```

Affichage sur le moniteur :

Liste chaînée : 30 -> 20 -> 10 -> NULL

Quelques explications :

1. Chaque **nœud** contient une **valeur** et un **pointeur** vers le prochain nœud.
2. `ajouterEnTete()` ajoute un élément en tête de liste.
3. `afficherListe()` parcourt la liste avec un pointeur.
4. `libererListe()` libère la mémoire allouée avec `delete`.

Déclaration et initialisation des pointeurs

1 - Déclaration d'un pointeur

Un pointeur est une variable qui stocke **l'adresse mémoire** d'une autre variable. Syntaxe : `type* nomPointeur;`

1. `type` : Le type de la variable pointée (ex: `int`, `double`).
2. `*` : Indique qu'il s'agit d'un pointeur.
3. `nomPointeur` : Nom du pointeur.

Exemple de déclaration :

```
int* ptr; // Pointeur vers un entier
double* dptr; // Pointeur vers un double
```

2 - Initialisation d'un pointeur

Un pointeur doit être **initialisé** avec l'adresse d'une variable existante à l'aide de `&` (opérateur d'adressage).

```
int a = 10;
int* ptr = &a; // ptr stocke l'adresse de a
```

Voici ce qu'on obtient en mémoire :

Nom	Valeur	Adresse en mémoire.
a	10	0x100
ptr	0x100	0x200

En résumé, `ptr` stocke `0x100`, qui est l'adresse de a.

3 - Déréférencement d'un pointeur

L'opérateur * permet d'accéder à la valeur stockée à l'adresse pointée (**déréférencement**).

```
std::cout << *ptr; // Affiche 10 (valeur de a)
```

!! **Attention** : Un pointeur non initialisé peut provoquer des erreurs (`segmentation fault`).

4 - Pointeur nul (`nullptr`)

Un pointeur peut être initialisé à `nullptr` pour éviter les **pointeurs suspendus**.

```
int* ptr = nullptr; // Pointeur ne pointant vers rien
```

Avant d'accéder à `*ptr`, il est conseillé de vérifier :

```
if (ptr != nullptr) {
  std::cout << *ptr; }
```

Exercices sur les Pointeurs en C++

Consigne : Pour les 5 programmes, **représentez l'illustration en mémoire** en indiquant :

1. **Les variables** et leur valeur
2. **Les pointeurs** et l'adresse qu'ils stockent
3. **Les accès mémoire** via **déréférencement (*)**

Exercice 1 : Déclaration et initialisation simple

```
#include <iostream>

int main() {
  int a = 5;
  int* ptr = &a;

  std::cout << "a = " << a << std::endl;
  std::cout << "*ptr = " << *ptr << std::endl;
  return 0;
}
```

→ Quelle est l'illustration mémoire après exécution ?

Nom	Valeur	Adresse mémoire

Exercice 2 : Modification via un pointeur

```
#include <iostream>

int main() {
  int x = 10;
  int* p = &x;

  *p = 20; // Modification via le pointeur

  std::cout << "x = " << x << std::endl;
  std::cout << "*p = " << *p << std::endl;
  return 0;
}
```

→ Décrivez l'état de la mémoire avant et après la modification.

Exercice 3 : Pointeur et Tableau

```cpp
#include <iostream>

int main() {
  int tab[3] = {1, 2, 3};
  int* p = tab;

  std::cout << "*p = " << *p << std::endl;
  std::cout << "*(p + 1) = " << *(p + 1) << std::endl;
  std::cout << "*(p + 2) = " << *(p + 2) << std::endl;

  return 0;
}
```

→ Donnez l'illustration mémoire et expliquez comment fonctionne l'arithmétique des pointeurs.

Exercice 4 : Allocation dynamique

```cpp
#include <iostream>

int main() {
  int* p = new int(42);

  std::cout << "*p = " << *p << std::endl;

  delete p; // Libération de la mémoire

  return 0;
}
```

→ Quel est l'état de la mémoire avant et après `delete` ?

Exercice 5 : Pointeurs et Fonctions

```
#include <iostream>

void modifier(int* p) {
  *p = 100;
}

int main() {
  int y = 50;
  modifier(&y);

  std::cout << "y = " << y << std::endl;

  return 0;
}
```

→ Expliquez comment la fonction `modifier()` modifie y et donnez la représentation mémoire.