LES POINTEURS

LES POINTEURS EN C++

Les **pointeurs** sont des variables qui contiennent l'adresse mémoire d'une autre variable.

Syntaxe :

```
Type_de_données *nom_du_pointeur;
```

Exemple d'utilisation :

```
int nombre = 42;
int *pointeur;
pointeur = &nombre;
```

Le symbole & permet de récupérer l'adresse mémoire d'une variable et de l'affecter au pointeur.

INTRODUCTION AUX POINTEURS

DÉFINITION D'UN POINTEUR

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

Les pointeurs peuvent être utilisés pour accéder à des variables de manière indirecte, ce qui permet de manipuler leur valeur sans les nommer directement.

ADRESSE MÉMOIRE

Dans un programme, chaque **variable** est stockée dans une **adresse mémoire** spécifique. Les **pointeurs** permettent de manipuler ces adresses directement.

UTILITÉ DES POINTEURS

- **Accéder** à des variables sans les connaître explicitement
- Passer des **arguments par référence** à une fonction
- Gérer la **mémoire dynamiquement**

DÉCLARER UN POINTEUR

SYNTAXE

```
type *nom;
```

Type est le type de la variable pointée et **nom** est le nom du pointeur.

Un pointeur stocke l'adresse mémoire d'une variable de son type déclaré.

EXEMPLES DE DÉCLARATION

```c
int *ptr_int;
double *ptr_double;
char *ptr_char;
```

Ces déclarations représentent des **pointeurs** vers des **entiers**, des **flottants** et des **caractères**, respectivement.

Les pointeurs sont des variables qui contiennent l'adresse mémoire d'autres variables. Ils sont couramment utilisés pour optimiser les opérations sur des données et gérer des structures de données complexes.

LES POINTEURS EN C++

OPÉRATIONS SUR LES POINTEURS

Les pointeurs peuvent être manipulés avec différentes **opérations**.

- **Assignation** : attribuer une adresse à un pointeur.
- **Déréférencement** : accéder à la valeur stockée à l'adresse indiquée par un pointeur.
- **Arithmétique des pointeurs** : ajouter ou soustraire des entiers à un pointeur pour accéder à d'autres éléments de la mémoire.

```
int a = 10;
int *p = &a;          // Assignation, p reçoit l'adresse de a
int b = *p + 2;       // Déréférencement, b vaut 12
p++;                 // Arithmétique, p pointe sur l'adresse mémoire suivante
```

AFFECTATION D'UNE ADRESSE

SYNTAXE

Pour affecter une adresse à un **pointeur** :

```
pointeur = &variable;
```

On utilise l'opérateur `&` pour obtenir l'adresse d'une variable.

EXEMPLES D'AFFECTATION

```
int x = 5;
int* ptr = &x;
```

Affectation d'une valeur à une variable :

- `int x = 5;` : on crée une variable `x` de type `int` et on lui affecte la valeur 5.

Création d'un pointeur :

- `int* ptr = &x;` : on crée un pointeur `ptr` de type `int` qui pointe vers l'adresse mémoire de la variable `x` (on utilise l'opérateur `&` pour obtenir l'adresse).

OPÉRATIONS ARITHMÉTIQUES SUR LES POINTEURS

ADDITION ET SOUSTRACTION

Il est possible d'**ajouter** ou **soustraire** un entier à un pointeur :

```
pointeur + n;
pointeur - n;
```

Les opérations d'addition ou de soustraction sur les pointeurs sont basées sur l'arithmétique des pointeurs, où le décalage dépend de la taille du type pointé.

EXEMPLES D'UTILISATION

```cpp
int tableau[5] = {1, 2, 3, 4, 5};
int* ptr = &tableau[0];

ptr = ptr + 2; // ptr pointe maintenant sur tableau[2]
```

Dans cet exemple, nous montrons comment manipuler un **pointeur** pour accéder à un élément d'un **tableau** en **C++**.

COMPARAISON DE POINTEURS

SYNTAXE

Les **pointeurs** peuvent être comparés avec les **opérateurs habituels** :

```
pointeur1 == pointeur2;
pointeur1 != pointeur2;
```

Les pointeurs stockent des adresses mémoire et leur comparaison est une comparaison des adresses et non de leur contenu.

EXEMPLES DE COMPARAISON

```
int x = 5, y = 10;
int* ptr1 = &x;
int* ptr2 = &y;

if (ptr1 != ptr2) {
  cout << "Les pointeurs sont différents." << endl;
}
```

Dans cet exemple, les pointeurs **ptr1** et **ptr2** pointent respectivement vers les variables **x** et **y**. La comparaison des pointeurs vérifie si les adresses mémoire pointées sont différentes.

ACCÈS AUX VARIABLES VIA LES POINTEURS

ACCÈS AUX VARIABLES VIA LES POINTEURS

Pour accéder à la variable pointée par un pointeur, on utilise l'**indirection (*)** et l'opérateur **flèche** (–>).

Opérateur	Description
*	Indirection
->	Flèche

- L'indirection (*) pemet d'accéder à la valeur de la variable pointée.
- La flèche (>) permet d'accéder aux membres d'une structure ou d'une union via un pointeur.

L'opérateur d'indirection est utilisé pour déréférencer un pointeur (obtenir la valeur pointée). L'opérateur flèche est utilisé spécifiquement pour les structures et les unions.

INDIRECTION

L'**indirection** consiste à accéder à la valeur de la variable pointée par un **pointeur**.

L'indirection peut être utilisée pour gérer efficacement la mémoire et permettre des manipulations plus avancées des données.

EXEMPLES D'UTILISATION

```
int a = 5;
int *p = &a; // p contient l'adresse de a
cout << *p;  // Affiche la valeur de a (5)
*p = 10;     // Modifie la valeur de a (10)
```

Les pointeurs sont des variables qui contiennent l'adresse d'une autre variable, permettant de manipuler directement sa valeur en mémoire.

OPÉRATEUR >

L'opérateur –> est utilisé pour accéder aux **membres** d'un **objet** pointé par un **pointeur**.

Syntaxe	Description
ptr->membre	Accède au membre membre de l'objet pointé par le pointeur ptr

SYNTAXE

```
Pointeur->membre
```

Cette syntaxe sert à accéder à un membre (variable, fonction) d'un objet via un pointeur. Il est équivalent à `(*pointeur).membre`.

EXEMPLES D'UTILISATION

```
class Person {
public:
  string name;
};
Person *p = new Person();
p->name = "Alice"; // Accès au membre `name` via le pointeur p
```

Dans cet exemple, nous créons un objet de la classe **Person** et utilisons le pointeur **p** pour accéder au membre **name** et y attribuer une valeur.

COMPARAISON AVEC L'OPÉRATEUR `.`

L'opérateur `->` est l'équivalent de l'opérateur `.` pour les **pointeurs**.

```
Person obj;
obj.name = "Alice"; // Accès au membre `name` via l'objet obj
```

POINTEURS ET TABLEAUX

RELATION ENTRE POINTEURS ET TABLEAUX

Les **pointeurs** et les **tableaux** ont une relation étroite en C++, car les tableaux sont en réalité des pointeurs vers le premier élément du tableau.

Pointeur	Tableau
int* ptr;	int arr[5];
ptr = arr;	// les deux sont équivalents
*ptr = arr[0];	// Accès au premier élément

Cette relation simplifie la manipulation des tableaux, mais il faut être prudent avec la gestion de la mémoire et les limites des tableaux.

ADRESSES DES ÉLÉMENTS DU TABLEAU

Chaque élément d'un tableau occupe un **emplacement mémoire**, dont l'adresse peut être obtenue à l'aide de l'**opérateur `&`**.

Index	Contenu	Adresse
0	élément0	&tableau[0]
1	élément1	&tableau[1]
2	élément2	&tableau[2]

UTILISATION DES POINTEURS POUR PARCOURIR UN TABLEAU

On peut parcourir un tableau en utilisant un **pointeur** et en **incrémentant** ou **décrémentant** son adresse.

Un pointeur est une variable contenant l'adresse mémoire d'une autre variable, permettant d'accéder à la valeur de celle-ci.

POINTEURS DE TABLEAU

DÉCLARATION

Pour déclarer un **pointeur de tableau**, on utilise la syntaxe suivante :

```
int (*ptr)[N];
```

N'oubliez pas d'expliquer que `N` doit être remplacé par la taille du tableau et que le type `int` peut être remplacé par d'autres types de données, selon le besoin.

EXEMPLE D'UTILISATION

```cpp
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
int (*ptr)[3] = arr;
```

Cet exemple montre comment déclarer un **pointeur** sur un tableau à deux dimensions en C++ et l'initialiser avec les valeurs du tableau `arr`.

POINTEURS SUR POINTEURS

DÉCLARATION

Un **pointeur sur pointeur** est déclaré en utilisant la syntaxe suivante :

```
int **ptr;
```

EXEMPLES D'UTILISATION

```
int x = 10;
int *p = &x;
int **q = &p;
```

Cette slide montre comment créer un pointeur sur un entier (p) et un pointeur sur un pointeur (q). La première ligne déclare un entier x, la seconde ligne crée un pointeur p contenant l'adresse de x, et la troisième ligne crée un pointeur q contenant l'adresse de p.

AVANTAGES ET INCONVÉNIENTS

- **Avantages** :
  - Permet de créer des tableaux **multidimensionnels dynamiques**
  - Facilite l'utilisation de certaines **structures de données**
- **Inconvénients** :
  - **Complexité accrue**
  - Risque d'**erreurs de mémoire** (fuites, accès incorrects)

POINTEURS ET FONCTIONS

PASSER DES POINTEURS EN ARGUMENTS

Les **pointeurs** peuvent être utilisés pour passer des **références** à des variables à une fonction, permettant ainsi une modification de la variable depuis la fonction.

```c
#include <stdio.h>

void modifierValeur(int *ptr) {
  *ptr = 10;
}

int main() {
  int x = 5;
  printf("Avant : %d\\n", x);
  modifierValeur(&x);
  printf("Apres : %d\\n", x);
  return 0;
}
```

Dans cet exemple, un pointeur est passé à la fonction "modifierValeur" qui permet de modifier la valeur de la variable "x" dans la fonction "main".

SYNTAXE

```
void maFonction(int *ptr);
```

Dans cette syntaxe, la fonction `maFonction` reçoit un argument sous forme de **pointeur** vers un entier. Les pointeurs sont utilisés pour accéder aux adresses mémoire et pour manipuler les données de manière efficace.

EXEMPLES D'UTILISATION

```cpp
#include <iostream>

void incrementer(int *ptr) {
    (*ptr)++;
}

int main() {
    int a = 5;
    incrementer(&a);
    std::cout << "a: " << a << std::endl; // a: 6
    return 0;
}
```

Cet exemple montre comment utiliser un **pointeur** pour manipuler la mémoire et modifier une variable directement dans la mémoire. Le pointeur est créé dans la fonction `incrementer()` et est utilisé pour incrémenter la valeur de la variable `a`.

AVANTAGES ET INCONVÉNIENTS

- **Avantage** : Modifie directement la **variable** sans avoir à renvoyer une nouvelle valeur.
- **Inconvénient** : Peut être moins **sécurisé** car la fonction a accès direct à l'adresse mémoire.

RETOURNER DES POINTEURS DEPUIS UNE FONCTION

Une fonction peut également retourner un **pointeur**, mais attention à ne pas retourner un pointeur vers une **variable locale**.

Les pointeurs vers les variables locales ne sont pas valides à l'extérieur de la fonction. Lorsque la fonction se termine, la mémoire allouée pour les variables locales est libérée. Si un pointeur vers une variable locale est retourné, il peut causer un comportement imprévisible. Utilisez des variables globales ou allouez explicitement de la mémoire pour éviter ce problème.

SYNTAXE

```
int *maFonction();
```

Cette fonction renvoie un pointeur vers un entier.

EXEMPLES D'UTILISATION

```cpp
#include <iostream>
#include <ctime>
#include <cstdlib>

int *creer_tableau_entiers(int taille) {
    int *tableau = new int[taille];

    for (int i = 0; i < taille; i++) {
        tableau[i] = rand() % 100;
    }

    return tableau;
}

int main() {
```

Cet exemple montre comment créer un tableau dynamique d'entiers, le remplir avec des nombres aléatoires, l'afficher et le libérer.

PRÉCAUTIONS ET ERREURS COURANTES

- **Ne pas retourner un pointeur** vers une variable locale car cette variable sera détruite à la fin de la fonction.
- Pensez à **libérer la mémoire** allouée avec `delete` pour éviter les fuites de mémoire.

GESTION DYNAMIQUE DE LA MÉMOIRE

GESTION DYNAMIQUE DE LA MÉMOIRE

La **gestion dynamique de la mémoire** permet d'**allouer** et de **libérer** de la mémoire à l'exécution d'un programme.

- Allocation de mémoire : `malloc`, `calloc`, `realloc`
- Libération de mémoire : `free`

`new` ET `delete`

`new` et `delete` sont des **opérateurs** utilisés pour **allouer** et **libérer** de la mémoire dynamiquement en C++.

- `new` permet d'allouer de la mémoire pour un objet ou un tableau d'objets
- `delete` permet de libérer la mémoire allouée précédemment

Opérateur	Utilisation	Exemple
new	Allouer de la mémoire dynamiquement	int* myInt = new int(42);
delete	Libérer la mémoire dynamiquement	delete myInt;

SYNTAXE

Allouer de la mémoire :

```
Type* ptr = new Type;
```

Libérer de la mémoire :

```
delete ptr;
```

EXEMPLES D'UTILISATION

Allouer un **entier** :

```
int* int_ptr = new int;
```

Allouer un **tableau d'entiers** :

```
int* arr_ptr = new int[10];
```

Libérer un **entier** :

```
delete int_ptr;
```

Libérer un **tableau d'entiers** :

```
delete[] arr_ptr;
```