



Chapitre 4: Pointeurs et structures de données

Dr. Khadhraoui Mayara

2025/2026

PLAN

1. Les pointeurs
2. Les structures de données

Section 1

Les pointeurs

- Mémoire d'un ordinateur $\approx$ ensemble de **cases-mémoire**. Chaque case est repérée par une **adresse**.

	Mémoire	Adresse
		0
		1
		2
		:
var	12	66844 (= &var)
		:

- L'adresse où est stockée la valeur d'une variable `var` s'obtient grâce à l'opérateur d'adresse `&` : cette adresse se note `&var`.

- **Pointeur** : variable contenant l'adresse d'une autre variable. La déclaration d'un pointeur est de la forme suivante :
- `<type> *<nom>;`

Par exemple :

```
int *p; // pointe sur une case quelconque
```

```
int x = 0;
```

```
cout << x << endl; // affiche 0
```

	Mém.	Adr.
p	58456	: 18424
x	0	: 36758
*p		: 58456

La variable `p` est alors un pointeur sur un `int`. La variable entière dont `p` contient l'adresse est dite pointée par `p` et se note `*p`.

- **Pointeur** : variable contenant l'adresse d'une autre variable. La déclaration d'un pointeur est de la forme suivante :

- `<type> *<nom>;`

Par exemple :

```
int *p; // pointe sur une case quelconque
```

```
int x = 0;
```

```
p = &x; // pointe maintenant sur x
```

```
cout << x << endl; // affiche 0
```

	Mém.	Adr.
p	36758	: 18424
*p,x	0	: 36758
		: 58456

La variable `p` est alors un pointeur sur un `int`. La variable entière dont `p` contient l'adresse est dite pointée par `p` et se note `*p`.

- **Pointeur** : variable contenant l'adresse d'une autre variable. La déclaration d'un pointeur est de la forme suivante :
- `<type> *<nom>;`

Par exemple :

```
int *p; // Déclaration d'un pointeur vers un entier (il peut contenir l'adresse d'un int)
```

```
int x = 0; // Déclaration d'une variable entière x, initialisée à 0
```

```
p = &x; // Le pointeur p reçoit l'adresse de x → p pointe maintenant vers x
```

```
*p = 5; // On accède à la case mémoire pointée par p (donc x) et on y met la valeur 5
```

```
cout << x << endl; // Affiche la valeur de x (qui vaut maintenant 5)  
puis passe à la ligne
```

	Mém.	Adr.
p	36758	: 18424
*p,x	5	: 36758
		: 58456

int *p;

- Déclare une **variable pointeur** p qui peut contenir l'adresse d'un int.
- **Type** : p est de type "pointeur vers int".
- Important : ici p **n'est pas initialisé** — il contient une adresse indéterminée (valeur aléatoire). Si on l'utilisait (*p) avant de l'initialiser, on risque un comportement indéfini (plantage).
- Conseil sûr : `int *p = nullptr;` pour l'initialiser proprement.

int x = 0;

- Crée une variable entière x et lui donne la valeur 0.
- En mémoire : une case contenant 0.
- `p = &x;`
- L'opérateur & signifie "adresse de". &x donne l'adresse mémoire où x est stocké.

2. Les pointeurs

- On affecte cette adresse à p. Maintenant p **pointe vers** x (c'est-à-dire qu'il contient l'adresse de x).
- Schéma mental : p ---> [case mémoire de x] (contient 0)

***p = 5;**

- * est l'opérateur de **déréférencement** : il accède à la valeur stockée à l'adresse contenue dans p.
- *p = 5; signifie donc : « aller à l'adresse contenue dans p (ici l'adresse de x) et y écrire la valeur 5 ».
- Effet : **la variable x devient 5.**

cout << x << endl;

- Affiche la valeur actuelle de x sur la console, puis passe à la ligne.
- Comme *p = 5 a modifié x, l'affichage sera 5.

Initialisation

Un pointeur s'initialise à nullptr.

```
double *ptr = nullptr; // aucune case mémoire n'est pointée
```

Types

Un pointeur est toujours lié au type sur lequel il pointe.

```
int *ip;
```

```
double *dp;
```

```
dp = ip; // illégal car ip et dp ne sont pas de même type
```

```
*dp = *ip; // correct (conversion implicite de type)
```

3. Les références

- Une **référence** est une variable qui coïncide avec une autre variable. La déclaration d'une référence est de la forme suivante :

`<type> &<nom> = <nom_var>;`

- Par exemple :

```
int n = 10;  
int &r = n; // r est une référence/alias sur n
```

n et r ont la **même adresse** : r peut être vue comme un alias de n.

```
r = 20; // modification de la valeur de n
```

- **Contrairement aux pointeurs**, une référence ne peut être initialisée qu'une seule fois : à la déclaration.
- Toute autre affectation modifie en fait la variable référencée.

3. Les pointeurs vs les références

Caractéristique	Pointeur (int *p)	Référence (int &r)
Contenu	Contient l' adresse d'une variable	Alias : pas une variable à part, juste un autre nom pour une variable existante
Accès à la valeur	*p	r directement
Déclaration	int *p = &x;	int &r = x;
Peut être nul	✓ Oui (p = nullptr;)	✗ Non, une référence doit toujours être liée à une variable
Peut changer de cible	✓ Oui (p = &y;)	✗ Non, r reste toujours lié à la même variable
Besoin d'opérateur *	✓ Oui pour accéder à la valeur	✗ Non, on utilise r directement
Allocation mémoire	Le pointeur est une variable indépendante	Pas de nouvelle case mémoire, partage la même que la variable originale
Modification de la valeur originale	✓ *p = 20;	✓ r = 20;

4. Passage des paramètres dans une fonction

Version naïve (passage par valeur) :

```
void swap(int a, int b) // ne fonctionne pas
{
    int aux; // variable auxiliaire servant pour l'échange

    aux = a;

    a = b;

    b = aux;
}
```

- Les lignes `aux = a;` `a = b;` `b = aux;` **échangent les copies a et b, pas x et y eux-mêmes.**
- Une fois que la fonction se termine, ces **copies disparaissent**, donc **x et y restent inchangés.**
- En C/C++, quand on passe un `int` à une fonction **par valeur**, la fonction ne reçoit qu'une **copie** de la variable.
- Tout changement sur cette copie **n'affecte pas la variable originale.**

4. Passage des paramètres dans une fonction

Version par références :

```
void swap(int &a, int &b) // style C++
{
    int aux; // variable auxiliaire servant pour l'échange

    aux = a;

    a = b;

    b = aux;
}
```

- **int &a** et **int &b** → a et b sont des alias des variables originales, donc tout changement affecte directement les variables appelantes.
 - **aux** sert juste à ne pas perdre la valeur de a pendant l'échange.
 - La fonction modifie **réellement les valeurs originales**, contrairement à la version “naïve” qui utilisait le passage par valeur.
- ➔ **Par référence** → échange directement les variables, simple et sûr.

4. Passage des paramètres dans une fonction

Version par pointeurs :

```
void swap(int *a, int *b) // style C
{
    int aux; // variable auxiliaire servant pour l'échange

    aux = *a;

    *a = *b;

    *b = aux;
}
```

- `int *a, *b` → `a` et `b` **contiennent des adresses** de variables, pas les valeurs elles-mêmes.
- `*a` et `*b` → permettent **d'accéder aux valeurs stockées aux adresses pointées**.
- L'échange fonctionne sur les **valeurs originales**, car on manipule **directement les adresses des variables**.

➔ **Par pointeur** → échange via les adresses, plus flexible mais plus compliqué à écrire.

5. Usage de & et *

&, quand utilisé avec une déclaration de variable (un type)

= **référence sur...**

```
int &ra = a; // ra est une référence sur a (un alias de a)
```

*, quand utilisé avec une déclaration de variable (un type)

= **pointeur sur...**

```
int *pa; // pa est un pointeur sur un entier
```

&, quand utilisé avec une variable déjà déclarée

= **adresse de...**

```
&a // adresse de la variable a
```

*, quand utilisé avec un pointeur déjà déclaré

= **valeur pointée par...**

```
cout << *pa << endl; // affiche la valeur pointée par pa
```

En C/C++ :

Le nom du tableau **T** n'est pas vraiment une variable ordinaire, il est un **pointeur constant vers le premier élément** du tableau.

Autrement dit :

$$T \equiv \&T[0]$$

T contient l'**adresse mémoire du premier élément** (T[0]).

Considérons :

```
int T[5]; // T est un tableau de 5 entiers
```

```
int *p = T;
```

Passage de tableau en paramètre de fonctions:

Il se fait par valeur, et :

- il n'y a pas de recopie du contenu du tableau (gain de temps)
- la fonction peut modifier le contenu du tableau

```
void annule(int *V, int n) // notation équivalente
// Déclare une fonction appelée annule
// int *V = pointeur vers le premier élément d'un tableau d'entiers
// int n = taille du tableau
// void = la fonction ne retourne rien
{
    for(int i = 0; i < n; i++) // boucle pour parcourir tous les indices de 0 à n-1
        V[i] = 0;             // met à 0 chaque élément du tableau
}
```

8. Variables dynamiques / statiques

Supposons déclaré :

```
int *p; // pointeur sur int
```

Pour l'instant, p ne pointe sur rien. Il faut réserver l'espace-mémoire nécessaire à un `int` par l'instruction:

```
p = new int; // new : opérateur d'allocation mémoire
```

Cette fois, p contient une véritable adresse, décidée par la machine pendant l'exécution.

On peut alors utiliser la variable pointée par p (qu'on nomme **variable dynamique** par opposition aux variables ordinaires qualifiées de **statiques**) :

```
*p = 12;
```

8. Variables dynamiques / statiques

La variable pointée par `p` n'est pas automatiquement désallouée en sortie de bloc (c'est l'intérêt). Lorsqu'on n'a plus besoin de la variable `*p`, il faut libérer l'espace-mémoire qu'elle occupe par l'instruction :

```
delete p; // delete : opérateur de désallocation mémoire
```

L'expression `*p` est dès lors illégale, jusqu'à une prochaine allocation par :

```
p = new int;
```

9. Gestion de la mémoire- variables dynamiques

Grâce aux pointeurs, on peut manipuler des **structures de données dynamiques** dont la taille n'est déterminée qu'au moment de l'exécution.

Supposons déclaré :

```
int *T;
```

L'allocation-mémoire nécessaire pour recevoir n entiers s'écrit :

```
T = new int[n]; // un tableau dynamique de n entiers
```

Intérêt : n n'est pas nécessairement une constante. La taille du tableau peut donc être décidée au moment de l'exécution, en fonction des besoins.

Désallocation :

```
delete [] T; // désallocation mémoire d'un tableau dynamique
```

10. Comprendre les pointeurs: exemple du double tableau

Une matrice 2D (double tableau) peut s'allouer dynamiquement. Il s'agit d'un tableau 1D où chaque case pointe vers un autre tableau 1D : **c'est un tableau de tableaux.**

On l'implémente comme un pointeur de pointeurs, donc :

```
int **T; // déclaration du double pointeur,  
        // il pointera sur le tableau 2D d'entiers
```

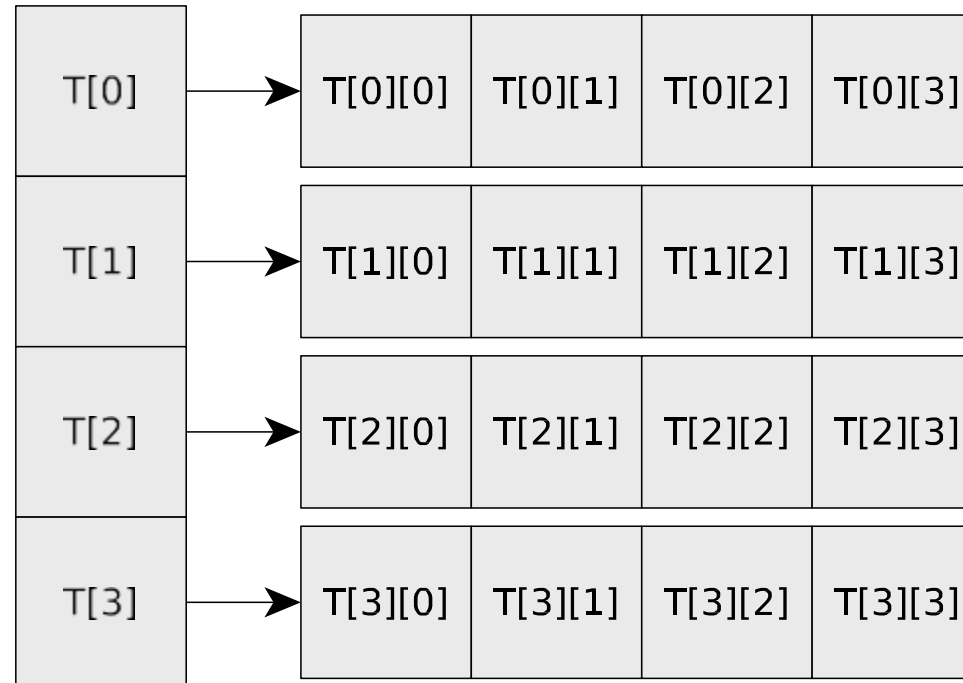
L'allocation-mémoire se déroule en deux temps :

```
T = new int*[nb_row]; // allocation du premier tableau : c'est un tableau de pointeurs sur  
des entiers (i.e. un tableau de tableaux d'entiers) ; le type de chaque cellule du tableau  
sera donc int*  
for(int i = 0; i < nb_row; ++i)  
  
    T[i] = new int[nb_col]; // allocation d'un tableau de nb_col éléments dans chaque case  
du premier tableau. Chaque élément de ce sous-tableau est de type int
```

Illustration de l'allocation dynamique :

```
int **T = new int*[nb_row];
```

```
for(int i = 0; i < nb_row; ++i)  
    T[i] = new int[nb_col];
```



Section 2

Les structures de données

1. Structures de données Séquentielles et linéaires

- Structures de données séquentielles (Tableaux)
- Structures de données linéaires (listes chaînées, piles, files)

Définition

*Un **tableau** est une structure de donnée T qui permet de stocker un certain nombre d'éléments $T[i]$ repérés par un index i . Les tableaux vérifient généralement les propriétés suivantes :*

- *tous les éléments ont le même type de base;*
- *le nombre d'éléments stockés est fixé;*
- *l'accès et la modification de l'élément numéro i est en temps constant $\Theta(1)$, indépendant de i et du nombre d'éléments, le tableau.*

NB: $\Theta(1)$: Cela signifie que **peu importe la position de l'élément dans le tableau (début, milieu ou fin)**, le temps nécessaire pour **accéder** à cet élément ou le **modifier** est **le même**.

Un tableau en mémoire

Définition

Dans le tableau, tous les éléments ont la même taille mémoire.

Nombre d'élément : n , taille d'un élément t :

$T[0]$	$T[1]$	$T[2]$	$\dots$	$T[n-1]$
--------	--------	--------	---------	----------

On suppose que le tableau commence à l'adresse d

- $T[0]$ occupe les cases d à $d + t - 1$;
- $T[1]$ occupe les cases $d + t$ à $d + 2t - 1$;
- $T[i]$ occupe les cases $d + it$ à $d + (i + 1)t - 1$;
- Le tableau entier occupe les cases d à $d + nt - 1$.

1. Les tableaux

Exemple concret :

Imaginons :

```
int T[4] = {10, 20, 30, 40};
```

- taille d'un int : $t = 4$ octets
- adresse de départ : $d = 1000$
- Etant donnée que: Les éléments sont consécutifs, espacés de 4 octets chacun.

Alors :

Élément	Calcul d'adresse	Cases mémoire occupées
T[0]	$d + 0 \times 4 = 1000$	1000 → 1003
T[1]	$d + 1 \times 4 = 1004$	1004 → 1007
T[2]	$d + 2 \times 4 = 1008$	1008 → 1011
T[3]	$d + 3 \times 4 = 1012$	1012 → 1015

Tableau partiellement remplis

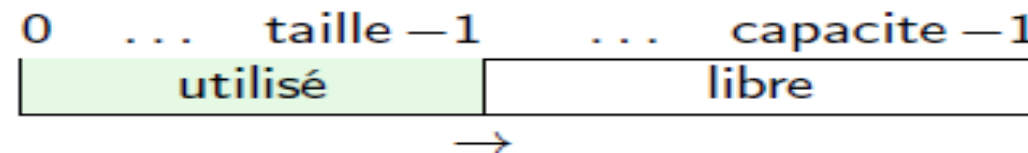
Retenir

Pour simuler un tableau de taille variable, on peut

- réserver une certaine quantité de mémoire appelée **capacité**,
- et ranger les valeurs **au début du tableau**.

Organisation des données (structure, classe) :

- tableau de taille **capacité** alloué
- éléments d'indice i pour $0 \leq i < \text{taille} \leq \text{capacité}$ initialisés



1. Les tableaux

Principe

Simuler un tableau de taille variable à partir d'un tableau fixe :

- Réserver une certaine capacité (mémoire allouée)
- Utiliser seulement les premières cases selon la taille actuelle

Organisation

Indice : 0 ... taille-1 →  utilisé

taille ... capacité-1 →  libre

Exemple 1

```
int tab[10]; // capacité = 10
```

```
int taille = 4;
```

```
tab[0]=12; tab[1]=7; tab[2]=5; tab[3]=9;
```

→ Tableau :

```
[ 12 | 7 | 5 | 9 | | | | | ]
```

↑ utilisés (0-3) ↑ libres (4-9)

1. Les tableaux

Exemple 2 – Ajout d'un élément

```
tab[taille] = 15;
```

```
taille++;
```

→ Nouveau tableau :

```
[ 12 | 7 | 5 | 9 | 15 |   |   |   |   ]
```

↑ utilisés (0–4)

À retenir

- capacité → espace total réservé
- taille → nombre d'éléments remplis
- Les valeurs sont rangées au début
- Permet de simuler un tableau dynamique simplement