

# Programmation C

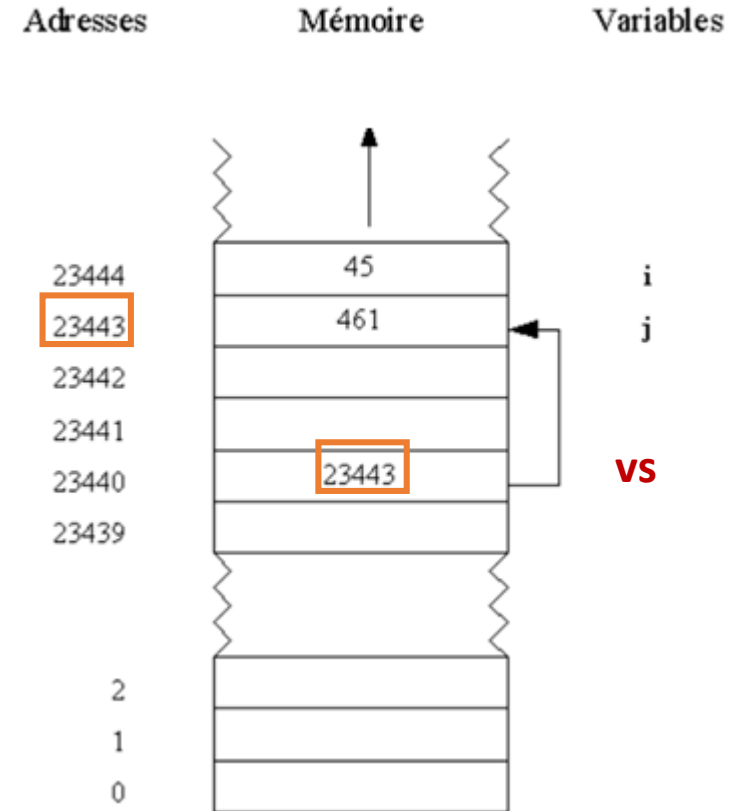
## Les pointeurs

Filière: GI, IDS – S1

A.U 2022-2023

# INTRODUCTION

- Toute variable manipulée dans un programme est stockée dans une partie de la mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiées par un numéro qu'on appelle adresse.
- Pour retrouver la case d'une variable il suffit connaître son **adresse** ou son **nom**. C'est pour cela on désigne les variables par des identificateurs.
- Le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Il est parfois très pratique de manipuler directement une variable par son adresse.



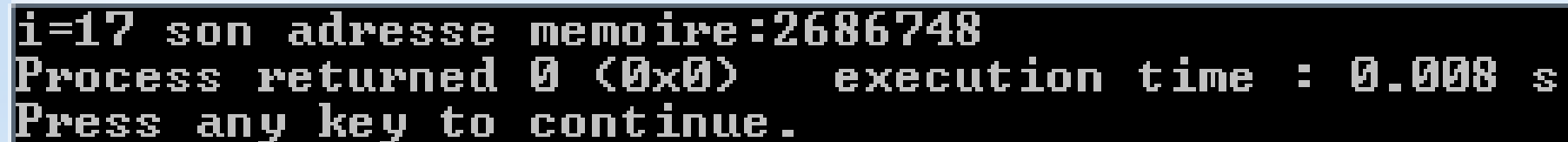
# Utilisation du pointeurs:

- Les langages de programmation (C,C++) offrent la possibilité de l'accès direct à la mémoire de l'ordinateur à l'aide des pointeurs.
  - On peut utiliser les pointeurs pour l'accès à une variable dans une fonction ou une composante d'un tableau...
- L'utilisation des pointeurs permet d'avoir accès à la couche basse de l'ordinateur, un accès direct à la mémoire. On peut littéralement se déplacer de case mémoire en case mémoire. Cette technique permet d'effectuer des optimisations sur l'utilisation de la mémoire ou la performance en terme de vitesse.

# 1. Les pointeurs

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i=17;
    printf ("i=%d son adresse memoire:%d", i,&i);
}
```

→ Résultat de l'exécution



```
i=17 son adresse memoire:2686748
Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

# 1. Les pointeurs

```
#include<stdio.h>
#include<stdlib.h>
int main()
] {
    int i=17,j=-32;

    printf ("i=%d son adresse memoire:%d", i,&i);
    printf ("j=%d son adresse memoire:%d\n", j,&j);

}
```

```
i=17 son adresse memoire:2686748
j=-32 son adresse memoire:2686744

Process returned 0 (0x0)    execution time : 0.013 s
Press any key to continue.
```

Nom variable			j	i		
Valeur			-32	17		
Adresse mémoire	....	.....	2686744	2686748	....	....

# 1. Les pointeurs

## Définition:

- **Variable:** sert à stocker les données manipulées par un programme. Lorsque l'on déclare une variable, un espace mémoire lui sera réservé pour y stocker sa valeur.  
L'emplacement de cet espace dans la mémoire est nommé adresse.
- **Pointeur:** *Un **pointeur** est une variable spéciale qui peut contenir l'**adresse** d'une autre variable.*

Nom variable		P		i		
Valeur		2686748		17		
Adresse mémoire	....	2686707		2686748	....	....



Si un pointeur P contient l'adresse d'une variable i, on dit que  
***'P pointe sur i'.***

# Type d'adressage

Adressage Direct (Passage par valeur)	Adressage indirect (Passage par adresse)
<ul style="list-style-type: none"><li>- Lorsqu'on accède à la case d'une variable à travers son nom (identificateur).</li><li>- C-à-d on change le contenu de variable à travers son nom.</li><li>- Ex: Annee = 2023;</li></ul>	<p>Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable i, nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée pointeur.</p> <p>Nous pouvons attaquer la case de variable i a travers ce pointeur P.</p>

# 1. Les pointeurs

## Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin:

- d'un opérateur 'adresse de': **&** pour obtenir l'adresse d'une variable.
- d'un opérateur 'contenu de': **\*** pour accéder au contenu d'une adresse.
- d'une syntaxe de déclaration pour pouvoir déclarer un pointeur.

Exemple:

Soit P un pointeur **non initialisé**, et A une variable (du même type) contenant la valeur 10 :  
A=10.

**P = &A;** // L'instruction suivante affecte l'adresse de la variable A à la variable P



# 1. Les pointeurs

## Déclaration:

Un pointeur est déclaré au moyen de l'opérateur de **déréférencement** «\*».

## Syntaxe :

**Type** \*Nom\_du\_Pointeur;

**Nom\_du\_Pointeur** ne peut (doit) recevoir que des adresses de variables du type **Type**

Exemple:

- `int a,b; // déclaration de deux variables de type entier`
- `int *ptr; // déclaration d'un pointeur sur une variable entière`
- `float *ptr; // déclaration d'un pointeur sur une variable float`
- `char nom, *ptr; // déclaration d'une variable de type char et un pointeur sur char`

# 1. Les pointeurs

Après les instructions,

**A=10;**

**→ P = &A;**

**B = \*P;**

**\*P = 20;**

- P pointe sur A,
- le contenu de A (référencé par \*P) est affecté à B, et
- le contenu de A (référencé par \*P) est mis à 20.

# 1. Les pointeurs

## Initialisation:

- Lorsqu'on déclare un pointeur **sans l'initialiser**, on ne sait pas sur quoi il pointe.
  - Pointeur qui ne pointe sur rien  
`int *ptr = NULL;`
  - Pointeur qui pointe sur une variable  
`int i=5;`  
`int *p=&i;`
  - Pointeurs qui pointent sur la même adresse  
`int i=5;`  
`int *p=&i;`  
`int *q=p;`

# 1. Les pointeurs

- **OPERATEURS ++ ET --**

Un pointeur peut **être déplacé** d'une adresse à une autre au moyen des opérateurs ++ et --.

L'unité d'incréméntation (ou de décréméntation) d'un pointeur est toujours **la taille de la variable pointée**.

**Exemple:**

```
#include <stdio.h>
main( )
{
    int i; /*i entier, supposons qu'il se trouve à l'adresse 100*/
    int *p; /*p pointeur sur un entier*/
    p=&i; /*p reçoit 100 : adresse de i*/
    p++; /*p s'incréménte de (2 ou 4) (devient 102 ou 104)*/
    char c; /*c char, supposons qu'il se trouve à l'adresse 200*/
    char *q; /*q pointeur sur char*/
    q=&c; /*q reçoit 200 : adresse de c*/
    q++; /*q s'incréménte de 1 (devient 201)*/
}
```

# 1. Les pointeurs

identificateur

contenu

Adresse

		P	
		0	
		2493527	

## Manipulation:

- Pointeur qui ne pointe sur rien: `int *p = NULL;`
- `int i=5;`
- `p=&i;`
- `printf("*p=%d\n",*p);` → `*p=?`

	i	P	
	?	?	
	2493523	2493527	

- `*p=80;`
- `printf("i=%d\n",i);` → `i=?`

	i	P	
	?	?	
	2493523	2493527	

- `int *q=p;`
- `printf("*q=%d\n",*p);` → `*q=?`
- `(*p)++;` // eq. à `i++;`

q	i	P	
?	?	?	
?	2493523	2493527	

# 1. Les pointeurs

## Manipulation:

- Pointeur qui ne pointe sur rien: `int *p = NULL;`
- `int i=5;`
- `p=&i;`
- `printf("*p=%d\n",*p);` → `*p=?`

- `*p=80;`
- `printf("i=%d\n",i);` → `i=?`

- `int *q=p;`
- `printf("*q=%d\n",*p);` → `*q=?`
- `(*p)++;` // eq. à `i++;`

		P	
		0	
		2493527	

	i	P	
	5	2493523	
	2493523	2493527	

	i	P	
	<b>80</b>	2493523	
	2493523	2493527	

q	i	P	
2493523	<b>80</b>	2493523	
2493513	2493523	2493527	

# 1. Les pointeurs

## Manipulation:

### Exercice:

Que produit le code suivant:

```
int a=51;  
int b=120;  
int * ptr;  
ptr = (a>b) ? &a : &b;  
(*ptr)++;  
printf("a=%d, b=%d, *pointeur=%d \n", a,b,*ptr);
```

**Le résultat:**

**a=51, b=121, \*pointeur=121**

# 1. Les pointeurs

## Rappel: Formats d'affichage de l'adresse:

- « **%d** »:pour afficher l'adresse en décimal.
- « **%x ou %X**»:pour affiche l'adresse en hexadécimal.
- « **%p** »:C'est un spécificateur réservée pour les pointeurs .

```
int a = 7;  
int *p = &a;  
printf("le pointeur p pointe sur l'adresse suivante: %p\n", p);  
printf("le pointeur p pointe sur l'adresse suivante: %x\n", p);  
printf("le pointeur p pointe sur l'adresse suivante: %d\n", p);
```

```
le pointeur p pointe sur l'adresse suivante: 000000000061FE14  
le pointeur p pointe sur l'adresse suivante: 61fe14  
le pointeur p pointe sur l'adresse suivante: 6422036
```

## Compléter les ... par les expressions correspondantes

- |                  |  |
|------------------|--|
| a désigne .....  | Le contenu de la variable a                        |
| &a désigne ..... | L'adresse de la variable a                         |
| p désigne .....  | L'adresse stockée dans le pointeur p               |
| *p désigne ..... | Le contenu de l'adresse stockée dans le pointeur p |



## 2. Arithmétique des pointeurs

Les opérations possibles sur les pointeurs :

Les seules opérations arithmétiques valides sur les pointeurs sont :

**1. L'addition d'un entier à un pointeur.** Le résultat est un pointeur de même type que le pointeur de départ.

**Pointeur + Entier = Pointeur**

**2. La soustraction d'un entier à un pointeur.** Le résultat est un pointeur de même type que le pointeur de départ.

**Pointeur - Entier = Pointeur**

**3. La différence de deux pointeurs** pointant tous deux vers **des objets de même type**. Le résultat est un entier.

**Pointeur - Pointeur = Entier**

**N.B:** la somme de deux pointeurs n'est pas autorisée.

## 2. Arithmétique des pointeurs

Si  $i$  est un entier et  $p$  est un pointeur sur un objet de type **TYPE**, l'expression  $p + i$  désigne un pointeur sur un objet de type **TYPE** dont la valeur est égale à la valeur de  $p$  incrémentée de  $i * \text{sizeof}(\text{TYPE})$ . Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentement et de décrémentation  $++$  et  $--$ .

$$P + i = P + i * \text{Sizeof}(\text{Type\_de\_pointeur})$$

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i = 3;
    int *p1, *p2;
    p1 = &i;
    p2 = p1 + 1;
    printf("p1 = %ld \t p2 = %ld\n", p1, p2);
    return 0;
}
```



```
p1 = 2293308      p2 = 2293312
Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

Démonstration de l'exemple:

On a:  $p1 = 2293308$

On sait que type de  $p$  est `int`

Donc  $\text{Sizeof}(\text{int}) = 4$

D'où :  $p1 + 1 = 2293308 + 1 * \text{Sizeof}(\text{int})$

$= 2293308 + 1 * 4$

$= 2293312$

## 2. Arithmétique des pointeurs

Par contre, le même programme avec des pointeurs sur des objets de type double :

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    double i = 3;
    double *p1, *p2;
    p1 = &i;
    p2 = p1 + 1;
    printf("p1 = %ld \t p2 = %ld\n", p1, p2);

    return 0;
}
```



```
p1 = 2293304    p2 = 2293312
Process returned 0 (0x0)    execution time : 0.110 s
Press any key to continue.
```

## 2. Arithmétique des pointeurs

Si  $p$  et  $q$  sont deux pointeurs sur des objets de type `Type`, l'expression  $p - q$  désigne un entier dont la valeur est égale à  $(p - q)/\text{sizeof}(\text{type})$ .

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i = 3, j = 1;
    int *p1, *p2, *p3;
    p1 = &i;
    p2 = &j;
    p3 = p1 - p2;
    printf("p1 = %ld \t p2 = %ld\n", p1, p2);
    printf("p3 = %ld \n", p3);
    return 0;
}
```



```
p1 = 2293300      p2 = 2293296
p3 = 1
```

## 2. Arithmétique des pointeurs

```
#include<stdio.h>;
main();
{
int i;
int *P;
P=&i;
printf("P=%x\n",P);
printf("P+1 =%x\n",P+1);
P++;
printf("nouvelle adresse de P = %x",P);
}
```

```
P=61fe14
P+1 =61fe18
nouvelle adresse de P = 61fe18
```

Différence entre P+1 et P++ ?

## 2. Arithmétique des pointeurs

### Incrémentation/Décrémentation

➤ Incrémentation du contenu de l'adresse via le pointeur:

Soit i une variable leur adresse stockée dans P

Pour incrémenter la valeur stockée dans i par le pointeur P:

$(*P)++ = i++ = i+1$

```
#include<stdio.h>;
main();
{
int i=1;
int *P;
P=&i;
printf(" i=%d \n ",i);
printf("*p=%d\n ",*P);
(*P)++;
printf("nouvelle valeur de i = %d",*P);
}
```

```
i=1
*p=1
nouvelle valeur de i = 2
```

Remarque: Pour exprimer la décrémentation P-- ou \*(p)-- on suivre le même principe.

# Exercice

En utilisant les pointeurs, écrire un programme qui **lit** et **affiche** les valeurs de 2 variables de type (int) et leurs adresses puis permute ces 2 entiers et affiche leurs adresses et leurs nouvelles valeurs à la fin.

# Solution

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int A,B,C,*P1,*P2,*P3;
    printf("veuillez entrer A : ");
    scanf("%d",&A);
    printf("veuillez entrer B : ");
    scanf("%d",&B);
    P1=&A;
    P2=&B;
    P3=&C;
    printf("A=%d\tB=%d\n",*P1,*P2);
    printf("P1 : %x\t P2 : %x \n",P1,P2);
    P3=P1;
    P1=P2;
    P2=P3;
    printf("A=%d\tB=%d\n",*P1,*P2);
    printf("NewP1 : %x\t NewP2 : %x \n",P1,P2);
}
```

```
veuillez entrer A : 1
veuillez entrer B : 2
A=1    B=2
P1 : 61fe04    P2 : 61fe00
A=2    B=1
NewP1 : 61fe00    NewP2 : 61fe04
```



## 2. Arithmétique des pointeurs

### Les opérateurs de comparaison

Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de **même type**.

L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux.

Exemple: le programme suivant **affiche** les éléments du tableau `tab` dans l'ordre croissant puis décroissant des indices.

# 2. Arithmétique des pointeurs

## Les opérateurs de comparaison

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int N=5;
    int tab[5] = {10, 20, 60, 0, 70};

    int *p;
    printf("\n Affichage par rdre croissant des indices:\n");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
        printf(" %d \n", *p);
    printf("\n Affichage par ordre decroissant des indices:\n");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
        printf(" %d \n", *p);
    return 0;
}
```

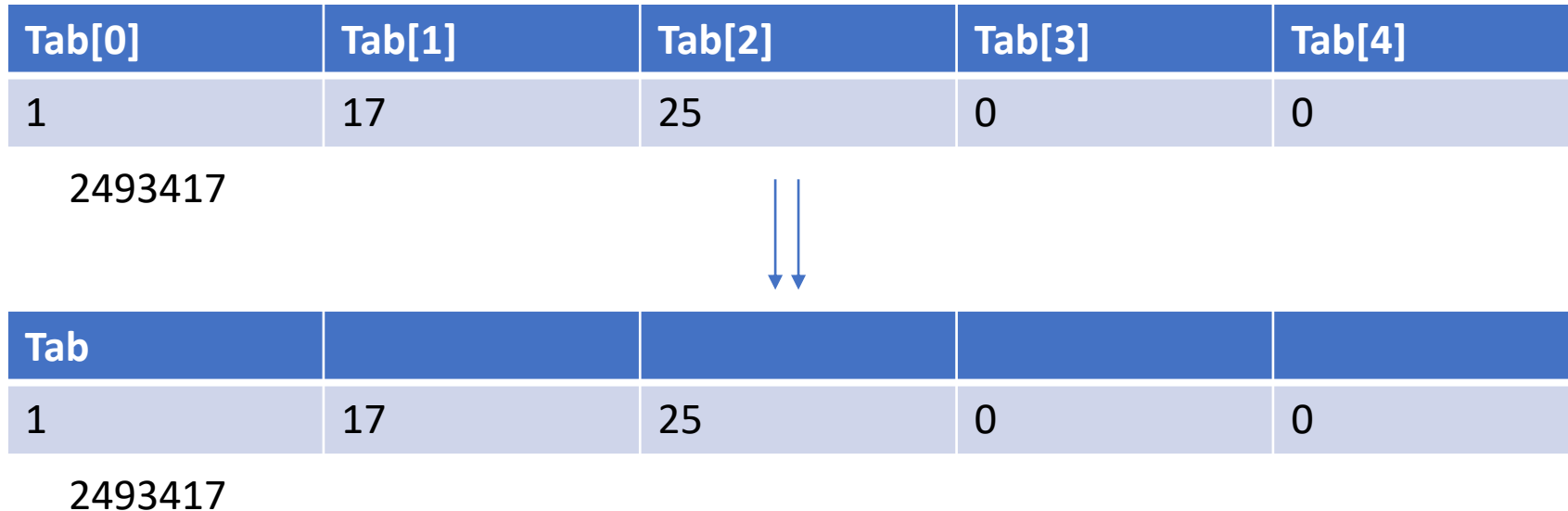
```
Affichage par rdre croissant des indices:
10
20
60
0
70

Affichage par ordre decroissant des indices:
70
0
60
20
10

Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

# 3. Les pointeurs et les tableaux

- `int Tab[5]={1,17, 25};`



- ✓ En C, Le nom d'un tableau représente l'adresse de son premier élément
- ✓ Équivalent à dire que le nom d'un tableau est un **pointeur constant** sur le premier élément du tableau

# 3. Les pointeurs et les tableaux

- Pointeurs et tableaux sont en étroite relation:

- **Exemple 1:**

```
int tab[10]={1,17, 25}, i;  
printf ("%d\n",*(tab)); // 1: tab pointe sur le 1er élément  
printf ("%d\n",*(tab+1)); // 17: tab+1 pointe sur le 2ème élément  
printf ("%d",*(tab+i)); // tab+i pointe sur le ième élément
```

- **Exemple 2:**

```
int tab[10]={1,58, 7}, i;  
int *ptr;  
ptr=tab; // équivalent à ptr=&tab[0];  
printf ("%d\n",*(ptr)); // 1: ptr pointe sur le 1er élément de tab  
printf ("%d\n",*(ptr+1)); // 58: ptr+1 point esur le 2ème élément de tab  
i=2;  
printf ("%d",*(ptr+i)); // 7 ptr+i point sur l'ième élément de tab
```

Tab				
1	17	25	0	0
2493417				

# 3. Les pointeurs et les tableaux

→ Exercice:

Déclarer un tableau de 5 éléments {5, -10, 15, 7, 25}

Afficher en utilisant une boucle **for** tous les éléments du tableau:

1. Utilisant les indexes
2. Utilisant les pointeurs

# 3. Les pointeurs et les tableaux

## Solution:

```
#include <stdio.h>

main()
{
    int tab[5]={5, -10, 15, 7, 25}, i;
    // Utilisant les indexes
    for (i=0; i<5; i++)
    {
        printf ("%d\t",tab[i]);
    }
    printf("\n");
    // Utilisant les pointeurs
    for (i=0; i<5; i++)
    {
        printf ("%d\t",*(tab+i));
    }
}
```

## 4. Sources d'erreurs

Un bon nombre d'erreurs lors de l'utilisation de C provient de la confusion entre soit contenu et adresse, soit pointeur et variable. Revoyons donc les trois types de déclarations que nous connaissons jusqu'ici et résumons les possibilités d'accès aux données qui se présentent.

### Les variables et leur utilisation

**int A;** déclare une variable simple du type int

A: désigne le contenu de A

&A: désigne l'adresse de A

**int B[];** déclare un tableau d'éléments du type int

B: désigne l'adresse de la première composante de B. (Cette adresse est toujours constante)

B[i]: désigne le contenu de la composante i du tableau

&B[i]: désigne l'adresse de la composante i du tableau

## 4. Sources d'erreurs

en utilisant le formalisme pointeur:

$B+i$ : désigne l'adresse de la composante  $i$  du tableau

$*(B+i)$ : désigne le contenu de la composante  $i$  du tableau

`int *P;` déclare un pointeur sur des éléments du type `int`.

`P` peut pointer sur des variables simples du type `int` ou sur les composantes d'un tableau du type `int`.

`P` désigne l'adresse contenue dans `P` (Cette adresse est variable)

`*P` désigne le contenu de l'adresse stockée dans `P`

Si `P` pointe dans un tableau, alors:

- `P` désigne l'adresse de la première composante
- `P+i` désigne l'adresse de la  $i$ -ième composante derrière `P`
- `*(P+i)` désigne le contenu de la  $i$ -ième composante derrière `P`



## 4. Sources d'erreurs

### **Exercice:**

**En utilisant la formalisme pointeur écrire un programme qui permet de copier les valeur positifs d'un tableau T1 dans un autre tableau T2.**

**La boucle parcourt les tableaux en utilisant les indexes.**

# Solution:

## Formalisme Tableau

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int tab[] = {-10, 20, -30, 40, -50, 60, -70, 80, -90, 100};
    int POS[10];
    int i, j;
    for (i=0, j=0; i<10; i++)
    {
        if (tab[i] > 0)
        {
            POS[j] = tab[i];
            j++;
        }
    }
    for (i=0; i<j; i++)
    {
        printf("POS[%d] = %d\n", i, POS[i]);
    }
}
```

Résultat

```
POS[0] = 20
POS[1] = 40
POS[2] = 60
POS[3] = 80
POS[4] = 100
```

## Formalisme Tableau

```
int main()
{
    int tab[] = {-10, 20, -30, 40, -50, 60, -70, 80, -90, 100};
    int POS[10];
    int i, j;
    for (i=0, j=0; i<10; i++)
    {
        if (*(tab+i) > 0)
        {
            *(POS+j) = *(tab+i);
            j++;
        }
    }
    for (i=0; i<j; i++)
    {
        printf("POS[%d] = %d\n", i, *(POS+i));
    }
    return 0;
}
```

→ Attention au parenthèses **\*(POS+i)**

```
for (i=0; i<j; i++)
{
    printf("POS[%d] = %d\n", i, *POS+i);
}
```

Que donne cette boucle?



```
POS[0] = 20
POS[1] = 21
POS[2] = 22
POS[3] = 23
POS[4] = 24
```

Programmation C  
Allocation dynamique

# Allocation dynamique de la mémoire

# Allocation dynamique de mémoire

## Taille d'une variable

- ✓ Pour toute variable créée, une zone mémoire lui sera associée, servant à stocker son contenu.
- ✓ La taille dépend du type de la variable (Selon l'architecture d'ordinateur):
  - char : 1 octet
  - int : 2 ou 4 octets (selon l'architecture du système)
  - float : 4 octets
  - double : 8 octets

# Allocation dynamique de mémoire

## ***Problème:***

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible. Il nous faut donc un moyen de gérer la mémoire lors de l'exécution du programme.

## ***Exemple***

***Tableau avec taille max***

# Allocation dynamique de mémoire

## Taille d'une variable:

- L'opérateur « sizeof() » retourne la taille en octets d'un type ou d'une variable passée en paramètre.
- sizeof( type) **ou bien** sizeof( nom\_variable)

- **Exemple:**

```
double x, tab[]={1,2,5,8};
```

```
printf("Sur mon système un 'double' fait %d octets", sizeof(x));
```

taille de la variable x: 8 octets

// équivalent à

- printf("Sur mon système un 'double' fait %d octets", sizeof(double));

→ taille de la variable du type double: 8 octets

# Allocation dynamique de mémoire

## Cas d'un pointeur:

- Contrairement aux variables, un pointeur n'a pas d'existence tant qu'on ne l'a pas initialisé.
- Il existe en C, des fonctions permettant d'allouer et de manipuler la mémoire à un pointeur.
  - malloc()
  - calloc()
  - realloc()
  - free()

# Allocation dynamique de mémoire

## ❑ **malloc():**

La fonction **malloc** de la bibliothèque « `stdlib` » permet de réserver de la mémoire **au cours d'exécution** d'un programme.

Syntaxe: **malloc(NombreOctets)**

- ✓ La fonction **malloc** de la bibliothèque `<stdlib>` nous aide à localiser et à réserver de la mémoire **au cours d'un programme**.
- ✓ fournit l'adresse d'un bloc en mémoire de `<N>` octets libres ou la valeur zéro s'il n'y a pas assez de mémoire.



# Allocation dynamique de mémoire

## Exemple 1: Sans allocation de mémoire

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptrChr;
    printf("Entrer un nombre: \n");
    scanf("%d",ptrChr);
    printf("Nombre saisi est: %d\n",*ptrChr);
    return 0;
}
```

**Erreur d'exécution: après la saisi du nombre par l'utilisateur, le programme va générer une erreur (Bug). Car le pointeur ptrChr n'a pas d'espace mémoire réservé.**

# Allocation dynamique de mémoire

## Exemple 2: Avec allocation de mémoire

```
#include <stdio.h>
#include<stdlib.h>
main()
{
    int *ptrChr;
    //Réservation de la mémoire
    ptrChr = malloc(255); /* lui réservé 255 octets en mémoire */
    printf("Entrer un nombre: \n");
    scanf("%d",ptrChr);
    printf("Nombre saisi est: %d\n",*ptrChr);
    return 0;}
```

→ Le nombre saisi sera affiché sans problème

# Remarque:

- Les pointeurs sont entre autre utilisés pour stocker les adresses des zones mémoires allouées **dynamiquement** par le programme. Si on alloue un bloc de taille  $T$  et qu'on reçoit l'adresse  $A$  en retour, cela veut dire que le bloc occupe la mémoire à partir de l'adresse  $A$  jusqu'à l'adresse  $A+T-1$ .

# Allocation dynamique de mémoire

## ❑ **calloc():**

La fonction `calloc` de la librairie `stdlib.h` a le même rôle que la fonction `malloc` mais elle initialise en plus l'objet pointé *\*p* à zéro.

**Syntaxe:** **`calloc(Nombre-Elements, Taille-Elements)`**

Ainsi, si `p` est de type `int*`, l'instruction

```
p = (int*)calloc(N,sizeof(int));
```

est strictement équivalente à:

```
p = malloc(N * sizeof(int));
```

```
for (i = 0; i < N; i++)
```

```
{ *(p + i) = 0; }
```

L'emploi de *calloc* est simplement plus rapide.

# Allocation dynamique de mémoire

## ❑ **realloc():**

realloc() s'utilise après qu'on ait utilisé la malloc() ou calloc(). Cette fonction permet de changer la taille d'un bloc de mémoire dynamique. Cela veut dire que si l'espace mémoire libre **qui suit le bloc à réallouer** est suffisamment grand, le bloc de mémoire est simplement agrandi. Par contre si l'espace libre n'est pas suffisant, un nouveau bloc de mémoire sera alloué, le contenu de la zone d'origine recopié dans la nouvelle zone et le bloc mémoire d'origine sera libéré automatiquement.

**Syntaxe:** **realloc(pointeur, memorySize)**

- Pointeur: l'adresse mémoire du bloc de mémoire à reallouer.
- memorySize : permet de spécifier la taille (en octets) du bloc de mémoire à allouer.

# Allocation dynamique de mémoire

## ❑ realloc(): Exemple

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int * p = malloc(20 * sizeof(int));
    if(p != NULL) {
        int I;
        for(I = 0; I < 20; I++) {
            p[I] = 1000 - (I * 10);
        }
        printf("Avant realloc : La valeur de p[15] = %i\n",p[15]);
        p = (int*) realloc (p, 21 * sizeof(int)); // Changement de la taille espace memoire
        if(p == NULL) {
            printf("Impossible de realloquer de la memoire dynamiquement !\n");
        } else {
            printf("Après ealloc : Valeur p[15] = %i\n",p[15]);
            free(p);
        }
    } else {
        printf("Impossible d'allouer de la memoire dynamiquement !\n");
    }
    return 0;
}
```

Avant realloc : La valeur de p[15] = 850

Après realloc : Valeur p[15] = 850

Process returned 0 (0x0) execution time : 0.009 s  
Press any key to continue.

# Allocation dynamique de mémoire

## ❑ Libération de la mémoire réservée `free()`:

- La fonction **free** de la biblio. « `stdlib` » permet la libération de l'emplacement mémoire réservé par `malloc` ou `calloc`.

### Syntaxe:

```
free(Pointeur);
```

**N.B:** A toute instruction de type `malloc` ou `calloc` doit être associée une instruction de type `free`.