

CHAPITRE 3

LES TABLEAUX ET LES POINTEURS

I. INTRODUCTION

I.1 TABLEAU

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës. La déclaration d'un tableau à une dimension se fait de la façon suivante :

type nom_du_tableau[nombre_d_éléments];

où nombre-éléments est une expression constante entière positive.

Par exemple, la déclaration **int tab[10];** indique que tab est un tableau de 10 éléments de type int. Cette déclaration alloue donc en mémoire pour l'objet tab un espace de 10×4 octets consécutifs.

Pour plus de clarté, il est recommandé de donner un nom à la constante nombre_d_éléments par une directive au préprocesseur, par exemple

#define nombre_d_éléments 10

On accède à un élément du tableau en lui appliquant l'opérateur []. Les éléments d'un tableau sont identifiés par leur position au sein de l'ensemble appelée **indice et chaque élément peut être manipulé comme une variable scalaire.**

Les éléments d'un tableau sont toujours numérotés de **0 à nombre_d_éléments -1**. Le programme suivant imprime les éléments du tableau tab :

```
#define N 10
main()
{
int tab[N];
int i;
...
for (i = 0; i < N; i++)
printf("tab[%d] = %d\n",i,tab[i]);
}
```

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est constant. Cela implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation. Par exemple, on ne peut pas écrire `tab1 = tab2;`. Il faut effectuer l'affectation pour chacun des éléments du tableau :

```
#define N 10
main()
{
int tab1[N], tab2[N];
int i;
...
for (i = 0; i < N; i++)
tab1[i] = tab2[i];
}
```

On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :
`type nom-du-tableau[N] = { constante1,constante2,...,constanteN};`

Par exemple, on peut écrire

```
#define N 4
main()
{
    int tab[N] = {1, 2, 3, 4};
}
```

Si le nombre de données dans la liste d'initialisation est inférieur à la dimension du tableau, seuls les premiers éléments seront initialisés. Les autres éléments seront mis à zéro si le tableau est une variable globale (extérieure à toute fonction) ou une variable locale de classe de mémorisation static.

De la même manière un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères littérale. Notons que le compilateur complète toute chaîne de caractères avec un caractère nul '\0'. Il faut donc que le tableau ait au moins un élément de plus que le nombre de caractères de la chaîne littérale.

```
#define N 8
char tab[N] = "exemple";
main()
{
    int i;
    for (i = 0; i < N; i++)
        printf("tab[%d] = %c\n",i,tab[i]);
}
```

Lors d'une initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de la liste d'initialisation. `char tab[] = "exemple";` De manière similaire, on peut déclarer un tableau à plusieurs dimensions. Par exemple, pour un tableau à deux dimensions :

type nom_du_tableau[nombre_lignes][nombre_colonnes]

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau. On accède à un élément du tableau par l'expression `tableau[i][j]`. Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

```
#include<stdio.h>
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};
main()
{
    int i, j;
    for (i = 0 ; i < M; i++)
    {
        for (j = 0; j < N; j++)
            printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
    }
}
```

```
#include<stdio.h>
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};
main()
{
    int i, j;
    printf("LES ELEMENTS DU TABLEAUX SONT\n");
    for (i = 0 ; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            printf("%d ",tab[i][j]);
        }
        printf("\n\n");
    }
}
```

I.2. LES POINTEURS

La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de pointeurs, c.-à-d. à l'aide de variables auxquelles on peut attribuer les adresses d'autres variables.

On peut donc définir un pointeur comme une *variable spéciale qui peut contenir l'adresse d'une autre variable*. En C, chaque pointeur est limité à un type de données.

Si un pointeur *P* contient l'adresse d'une variable *X*, on dit que

'P pointe sur X'.

Attention : Les *pointeurs* et les *noms de variables* ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :

- ✓ Un pointeur est une variable qui peut 'pointer' sur différentes adresses.
- ✓ Le nom d'une variable reste toujours lié à la même adresse.

DECLARATION D'UN POINTEUR

type * nom_pointeur ;

int *ptr ;

peut être interprétée comme suit:

"*ptr est du type int" ou "ptr est un pointeur sur int" ou

"ptr peut contenir l'adresse d'une variable du type int"

I.2.1 LES OPERATIONS ELEMENTAIRES SUR POINTEURS

En travaillant avec des pointeurs, nous devons observer les règles suivantes:

Priorité de * et &

- Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incréméntation ++, la décréméntation --). Dans une même expression, les opérateurs unaires *, & ,! ++, -- sont évalués de droite à gauche.
- Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple

Après l'instruction **P = &X;**

les expressions suivantes, sont équivalentes:

Y = *P+1	↔	Y = X+1
*P = *P+10	↔	X = X+10
*P += 2	↔	X += 2
++*P	↔	++X
(*P)++	↔	X++

int *Ptr ; Ptr est un pointeur sur un entier

int entier ; entier est une variable de type entier ;

Ptr=&entier ; Ptr pointe sur la variable entier

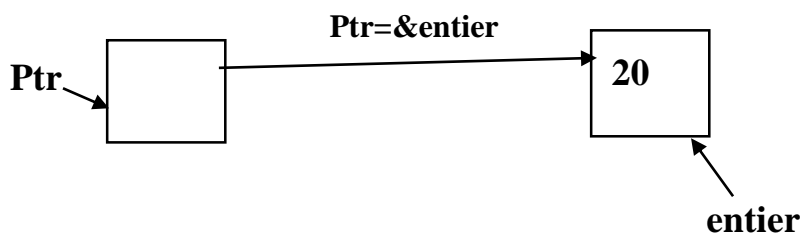
***Ptr :** Contenu de la zone pointée par Ptr

int entier=20 ;

int *ptr ;

Ptr=entier ;

Ptr=&entier ;



```
#include<stdio.h>
main()
{
    int entier=20;
    int *Ptr;
    Ptr=&entier;
    printf("L'adresse de la variable entier est:%x\n",&entier);
    printf("Le contenu de Ptr est :%x\n",Ptr);
    printf("La valeur de la variable entier est:%d\n",entier);
    printf("Le contenu de la zone pointee par Ptr est :%d\n",*Ptr);
}
```



I.2.2 ADRESSAGE DES COMPOSANTES D'UN TABLEAU

Comme nous l'avons déjà constaté, le nom d'un tableau représente l'adresse de son premier élément. En d'autre termes:

&tableau[0] et **tableau** sont une seule et même adresse.

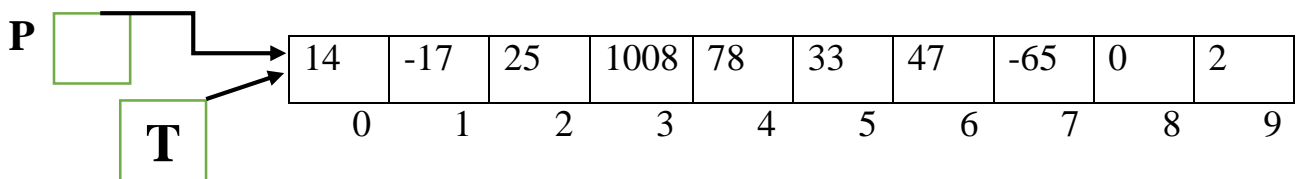
En simplifiant, nous pouvons retenir que le nom d'un tableau est un pointeur constant sur le premier élément du tableau, par exemple :

En déclarant un tableau **T** de type *int* et un pointeur **P** sur *int*,

int T[10];

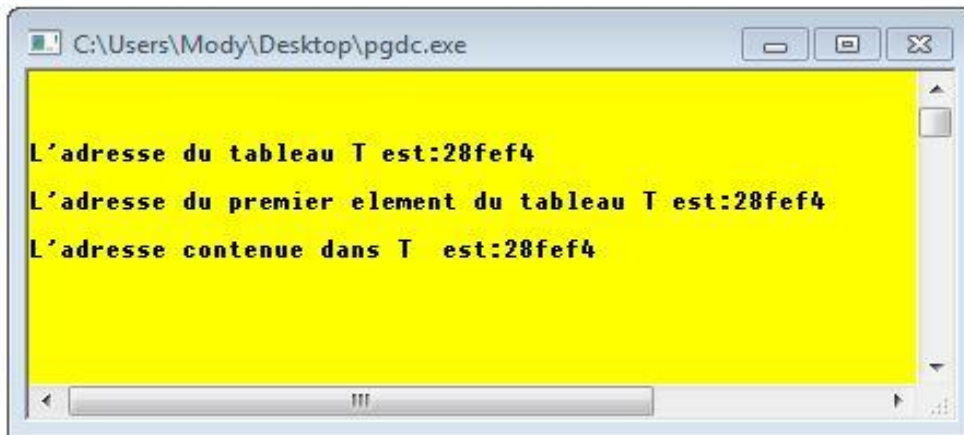
*int *P;*

p=T ; équivaut à P = &T[0]



Exemple :

```
#include<stdio.h>
#include<conio.h>
main()
{
    int T[10]={ 14,-17,25,1008,78,33,47,-65,2,0};
    int *P;
    P=T;
    printf("\n\nL'adresse du tableau T est:%x\n\n",T);
    printf("L'adresse du premier element du tableau T est:%x\n\n",&T[0]);
    printf("L'adresse contenue dans T est:%x\n\n",P);
    getch();
}
```



```
C:\Users\Mody\Desktop\pgdc.exe

L'adresse du tableau T est:28fef4
L'adresse du premier element du tableau T est:28fef4
L'adresse contenue dans T est:28fef4
```

Si **P** pointe sur une composante quelconque d'un tableau, alors **P+1** pointe sur la composante **suivante**. Plus généralement,

P+i

pointe sur la **i-ième** composante **derrière P** et

P-i

pointe sur la **i-ième** composante **devant P**.

Ainsi, après l'instruction,

P = T;

le pointeur **P** pointe sur **T[0]**, et

***(P+1)**

désigne le contenu de **T[1]**

...

***(P+i)**

désigne le contenu de **T[i]**