# Structures et Pointeurs

## I. Les structures

Une *structure* est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé *membre* ou *champ*, est désigné par un identificateur.

### 1. Déclaration

```
struct ma_structure{
        type_1  var1;
        type_2  var2;
         ……
        type_n  varn;
}
```

Dans cette définition, le mot clé **struct** fait partie du nom du type et ma_structure nous permettra de définir des variables de ce type. Ce dernier doit suivre les règles de constructions des variables.

### 2. Définition d'une variable d'un type structuré.

Maintenant qu'on a notre type structuré, on peut en définir des variables.
Cela se fera comme avec les variables des autre types.( Ne pas oublier le mot struct)
**struct personne** p1, p2, p[100];

### 3. Accès aux champs de la structure

Cela se fera avec l'opérateur « . » qu'il faudra mettre après le nom de la variable puis le faire suivre par le nom du champ.

Exemple

```
struct complexe
{
  double reelle;
```

```
  double imaginaire;
};
  struct complexe z;
  double norme;
  norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
  printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
```

En ANSI C, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux). Dans le contexte précédent, on peut écrire :

```
struct complexe z1, z2;
  z2 = z1;
```

### 4. Initialisation d'une structure

Le langage C nous permet d'initialiser une structure entière en une seule affectation, de la manière qui suit :

```
struct complexe z = {2,2};
```

Notez toutefois que ce type d'affectation ne peut se faire qu'à la déclaration, et nulle part ailleurs. De plus, en utilisant cette syntaxe, vous devez initialiser tous les champs de la structure.

### *5.* Tableaux de structures

Etant donné qu'une structure est composée d'éléments de taille fixe, il est possible de créer un tableau ne contenant que des éléments du type d'une structure donnée. Il suffit de créer un tableau dont le type est celui de la structure et de le repérer par un nom de variable :

***struct Nom_Structure Nom_Tableau[Nb_Elements];***

Chaque élément du tableau représente alors une structure du type que l'on a défini...
Le tableau suivant (nommé tab_complexe) pourra par exemple contenir 10 variables structurées de type *struct complexe* :

*struct complexe tab_complexe[10];*

De la même façon, il est possible de manipuler des structures dans les fonctions

- Le nom des champs répond aux critères des noms de variable
- Deux champs ne peuvent avoir le même nom

- Les données peuvent être de n'importe quel type hormis le type de la structure dans laquelle elles se trouvent

Ainsi, la structure suivante est correcte :

```
struct MaStructure {
        int Age;
        char Sexe;
        char Nom[12];
        float MoyenneScolaire;
        struct AutreStructure StructBis;

/* en considérant que la structure AutreStructure est définie */
};
```

## II. Les Pointeurs

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle *adresse*. Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets). Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

### 1. *L'importance des pointeurs en C*

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*.

En C, les pointeurs jouent un rôle primordial dans la définition de fonctions: Comme le passage des paramètres en C se fait toujours par la valeur, les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions. Ainsi le traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs (voir Chapitre 10).

En outre, les pointeurs nous permettent d'écrire des programmes plus compacts et plus efficients et fournissent souvent la seule solution raisonnable à un problème. Ainsi, la majorité des applications écrites en C profitent extensivement des pointeurs.

Syntaxe:

Type_variable_pointée * nom_variable_pointeur ;

Cette déclaration crée un pointeur *nom_variable_pointeur* vers une variable de type *type_variable_pointée*.

Exemple:

char *p_char;

int *p_int ;

L'opérateur * est utilisé pour dire que les variables p_char et p_int contiendront les adresses de variables de type char et int.

**2. Initialisation**

Puisque lors de la définition d'un pointeur, la valeur n'est pas connue, il faut l'initialiser.

Pour cela, il suffit simplement de lui assigner une valeur de même type, c'est-à-dire une adresse vers une variable pointée.

Exemple:

int i; char c;

int *p_int = &i; char *p_char = &c;

Dés lors, les pointeurs p_int et p_char pointent vers les variables i et c, c'est-à-dire qu'ils contiennent les adresses de début de ces variables.

**3. Manipulation**

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.

### 3.1 L'opérateur 'adresse de' : &

**&<NomVariable> fournit l'adresse de la variable <NomVariable>**

L'opérateur **&** nous est déjà familier par la fonction **scanf,** qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

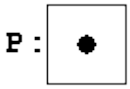
***Exemple***

```
int N;
printf("Entrez un nombre entier : ");
scanf("%d", &N);
```

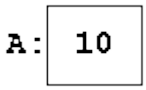
***Attention !*** L'opérateur **&** peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c.-à-d. à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

***Représentation schématique***

Soit P un pointeur non initialisé



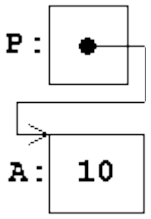

et A une variable (du même type) contenant la valeur 10 :




Alors l'instruction

```
P = &A;
```

affecte l'adresse de la variable A à la variable P. En mémoire, A et P se présentent comme dans le graphique à la fin du chapitre 9.1.2. Dans notre représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche:



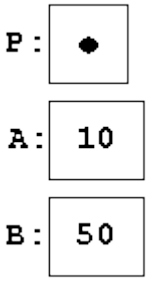
### *3.2 L'opérateur 'contenu de' : **

***<NomPointeur>***

désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>

***Exemple***

Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé:



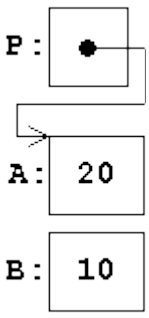

Après les instructions,

```
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.




## III. Pointeurs et Tableaux

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*.

***Exemple***

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

***int A[10];***
***int *P;***

L'instruction:

**`P = A;`** est équivalente à **`P = &A[0];`**


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 = A;**

le pointeur P pointe sur A[0], et

```
*(P+1) désigne le contenu de A[1]
*(P+2) désigne le contenu de A[2]
...    ...
*(P+i) désigne le contenu de A[i]
```

**Allocation dynamique de la mémoire**

malloc. (stdlib.h, malloc.h, alloc.h)
Cette fonction permet de réserver un bloc mémoire et de pouvoir y accéder à l'aide du pointeur retourné par cette fonction.
malloc prend alors à la volée un bloc mémoire donné par la taille définie.

**Syntaxe :**
ptr = malloc( taille_en_octet_de_la_zone_mémoire_souhaitée );
Dans cette définition, ptr sera donc le pointeur qui recevra l'adresse de la zone allouée.

**Exemple** :

```
int * t;
t = malloc(20);
```

Alloue un espace mémoire de 20 Octets et met l'adresse de cet espace dans le pointeur t.
On crée ainsi un tableau d'entiers dont on pourra accéder à ses éléments via t[i].
On peut également écrire `t = malloc(20*sizeof(int));`

Il peut arriver que la fonction malloc ne puisse allouer l'espace demandé pour une telle ou telle autre raison. Dans ce cas, elle renvoie un pointeur nul.
Avant tout traitement sur un pointeur renvoyé par malloc, vérifier qu'il n'est pas nul.

**Exemple** :

```
int * t = malloc(20*sizeof(int));
if(t = = null)  printf(''L'allocation ne s'est pas effectuée correctement!'');
else {
        ……
      }
```

La fonction malloc alloue un espace et renvoie un pointeur sur cet espace indépendamment du type de données ultérieurement stocké dans celui-ci.
Apres l'allocation donc, malloc renvoie un pointeur de type void*.
Cela veut dire que ce pointeur peut référencer n'importe quel type de données
Pour une conformité formelle et pour une portabilité, nous convertirons le pointeur renvoyé par malloc au type affecté.

```
int *t ;
t=(int*)malloc(20*sizeof(int));
float *f;
f=(float*)malloc(20*sizeof(float));
type * ptr;
ptr=(type*)malloc(20*sizeof(type));
```

Bien qu'elles soient moins fondamentales que les précédentes, les deux fonctions calloc et realloc peuvent s'avérer pratiques dans certaines circonstances.

La fonction calloc alloue l'emplacement nécessaire à nb_blocs consécutifs, ayant chacun une taille de taille octets.
Contrairement à ce qui se passait avec malloc, où le contenu de l'espace mémoire alloué était imprévisible, calloc remet à zéro chacun des octets de la zone ainsi allouée.

La taille de chaque bloc, ainsi que leur nombre sont tous deux de type size_t. On voit ainsi qu'il est possible d'allouer en une seule fois une place mémoire (de plusieurs blocs) beaucoup plus importante que celle allouée par malloc (la taille limite théorique étant maintenant size_t*size_t au lieu de size_t).

**void * calloc ( size_t nb_blocs, size_t taille )**

La fonction **realloc** permet de modifier la taille d'une zone préalablement allouée (par malloc, calloc ou realloc).

Le pointeur doit être l'adresse de début de la zone dont on veut modifier la taille. Quant à taille, de type size_t, elle représente la nouvelle taille souhaitée. Cette fonction restitue l'adresse de la nouvelle zone ou un pointeur nul dans le cas où l'allocation a échoué. Lorsque la nouvelle taille demandée est supérieure à l'ancienne, le contenu de l'ancienne zone est conservé (quitte à le recopier si la nouvelle adresse est différente de l'ancienne). Dans le cas où la nouvelle taille est inférieure à l'ancienne, le début de l'ancienne zone (c'est-à-dire taille octets) verra son contenu inchangé.

**void * realloc (void * pointeur, size_t taille )**

## IV. Pointeurs et fonctions

Les passages des paramètres se font par valeur.
On passe une copie de la variable prise comme paramètre à la fonction. La fonction ne peut pas changer la valeur de la variable.

**Exemple** :

```
#include<stdio.h>
int i=4;
void affect(int x){x=x+1;}
main(){
affect(i);
printf("%d",i);
}
```

On peut souhaiter que la fonction puisse changer la valeur d'un paramètre, pour cela, il faut passer ce paramètre par adresse (utilisation des pointeurs).
Utiliser les opérateurs d'adressage * et &;

Exemple :

```
#include<stdio.h>
int i=4;
void affect(int *x){*x=*x+1;}
main(){
```

```
affect(&i);
printf("%d",i);
}
```

## La fonction free

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, alors nous pouvons le libérer à l'aide de la fonction **free** de la bibliothèque <*stdlib*>. ***free( <Pointeur> )*** //libère le bloc de mémoire désigné par le <Pointeur>; n'a pas d'effet si le pointeur a la valeur zéro.