1- 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 contigües en mémoire. Chaque élément de la structure, appelé *membre* ou *champ,* est désigné par un identificateur.

La **déclaration d'un modèle de structure** dont l'identificateur est *identificateur1* suit la syntaxe suivante :

```
struct   identificateur1
{
type1 membre-1;
type2 membre-2;
...
type-n membre-n;
};
```

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe :

```
struct identificateur1 objet;
```

ou bien, si identificateur1 n'a pas et déclaré au préalable :

```
struct identificateur1
{
type-1 membre-1;

type-2 membre-2;
...
type-n membre-n;
} objet;
```

On accède aux membres d'une structure grâce à l'opérateur *membre de structure*, noté "**.**" i-ème membre de l'*objet* qui est désigné par l'expression

```
objet.membre-i
```

On peut effectuer sur le i-ème membre de la structure toutes les opérations valides sur des données de type *type-i*.

Par exemple, le programme suivant définit la structure complexe, composée de deux champs de type double ; il calcule la somme de deux nombres complexes.

```
#include <stdio.h>

struct complexe
{
double reelle;
double imaginaire;
};

main()
{
struct complexe z1 ={1., 2.};
struct complexe z2 ={10., 2.};
struct complexe RES;

RES.reelle = z1.reelle + z2.reelle ;
RES.imaginaire = z1.imaginaire + z2.imaginaire ;
printf("la somme de z1 + z2 = %lf+%lf i \n",RES.reelle,RES.imaginaire);
}
```

2- Définition avec typedef

Pour simplifier l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de typedef :

typedef ***type synonyme***;

Par exemple,

```
struct complexe
{
double reelle;
double imaginaire;
};
typedef struct complexe complexe;

main()
{
complexe z;
...
}
```

3- Les champs de bits

➔ On peut spécifier la longueur des champs d'une structure si ce champ est de type entier. Cela se fait en précisant le nombre de bits du champ avant le ; qui suit sa déclaration.

Par exemple,

```
struct CH1
{
unsigned int valeur1 : 1;
unsigned int valeur2 : 31;
};
```

valeur1 → codé sur un seul bit, et valeur2 → codé sur 31 bits.

Tout objet de type struct CH1 est codé sur 32 bits.

➔ l'ordre dans lequel les champs sont placés à l'intérieur de ce mot de 32 bits dépend de l'implémentation.

➔ Le champ valeur1 de la structure ne peut prendre que les valeurs 0 et 1.

4- LES UNIONS

➔ Une *union* désigne un ensemble de variables de types différent susceptibles d'occuper *alternativement* une même zone mémoire.

➔ Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

➔ Les déclarations et les opérations sur les objets de type union sont les mêmes que celles sur les objets de type struct.

Exemple :

```
union jour
{
char lettre;
int numero;
};

main()
{
union jour hier, demain;
hier.lettre = 'J';
printf("hier = %c\n",hier.lettre);
hier.numero = 4;
demain.numero = (hier.numero + 2) % 7;
printf("demain = %d\n",demain.numero);
}
```

Les unions peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents.

5- LES ENUMERATIONS

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre.

➔ Un objet de type énumération est défini par le mot-clef **enum** et un identificateur de **ID**, suivis de la liste des valeurs que peut prendre cet objet :

enum ***ID*** *{constante-1, constante-2,...,constante-n};*

➔ les objets de type enum sont représentés comme des int.

➔ Les valeurs possibles *constante-1, constante-2,...,constante-n* sont codées par des entiers de 0 à *n-1*.

Par exemple, le type enum booleen défini dans le programme suivant associe l'entier 0 à la valeur faux et l'entier 1 à la valeur vrai.

```
main()
{
enum booleen {faux, vrai};
enum booleen b;
b = vrai;
printf("b = %d\n",b);
}
```

➔ On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple :

enum booleen {faux = 13, vrai = 07};

6- POINTEURS ET STRUCTURES

6.1- Pointeur sur une structure

➔ On peut manipuler des pointeurs sur des structures. Le programme suivant crée, à l'aide d'un pointeur, un tableau d'objets de type structure.

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

struct eleve
{
char nom[20];
int date;
};
typedef struct eleve *classe;

main()
{
int n, i;
classe tab;

printf("nombre d'eleves de la classe = ");
scanf("%d",&n);

tab = (classe)malloc(n * sizeof(struct eleve));

for (i =0 ; i < n; i++)
{
printf("\n saisie de l'eleve numero %d\n",i);
printf("nom de l'eleve = ");
scanf("%s",&tab[i].nom);

printf("\n date de naissance JJMMAA = ");
scanf("%d",&tab[i].date);
}
```

```
printf("\n Entrez un numero      ");
scanf("%d",&i);
printf("\n Eleve numero %d:",i);
printf("\n nom = %s",tab[i].nom);
printf("\n date de naissance = %d\n",tab[i].date);
free(tab);
}
```

Si p est un pointeur sur une structure, on peut accéder à un membre de la structure pointé par l'expression

(*p).*membre*

➔ L'usage de parenthèses est ici indispensable car l'opérateur d'indirection * à une priorité plus élevée que l'opérateur de membre de structure.

Cette notation peut être simplifiée grâce à l'opérateur *pointeur de membre de structure*, noté **->**. L'expression précédente est strictement équivalente à

p->*membre*

➔ Ainsi, dans le programme précédent, on peut remplacer **tab[i].nom** et **tab[i].date** respectivement par **(tab + i)->nom** et **(tab + i)->date**.