

## Structures et Pointeurs

### I Les structures

#### 1. Définition

Ensemble de plusieurs objets de types éventuellement différents, regroupés sous un même nom.

Exemples de structure :

En mathématiques, un vecteur A peut être vu comme une structure dont les composantes x, y, etc. sont les différents champs.

En physique/chimie, les atomes dans le tableau de Mendeleïev sont des structures dont les propriétés (masse atomique, nombre d'Avogadro, etc.) sont les champs.

#### 2. Déclaration d'une structure

Syntaxe :

```
struct <identificateur>{  
    declar1 ;  
    declar2 ;  
    ...  
    declark ;  
};
```

Où `declari` est la déclaration du champ n°i de la structure.

```
struct date {  
    int jour ;  
    int mois ;  
    int annee ;  
    char nom_mois[9] ;  
};
```

Pour déclarer alors des variables (ou objets) de type structuré, on peut écrire

```
struct date x,y,p ;
```

x,y,p sont des variables de structure date

On peut aussi combiner :

- déclaration de structure
- et déclaration d'objets structurés

Exemple :

```
struct date {  
    int jour ;  
    int mois  
    int annee ;  
    char nom_mois[9] ;  
} x,y,p ;
```

On peut initialiser directement une variable de type structuré.

Exemple :

```
struct date x={24,2,1989, « MARS »}
```

### 3. Accès à un champ d'une variable structurée

<Identificateur variable structurée>.<identificateur champ>

Exemple :

z=x.annee ;     renvoie la valeur entière de la variable x de structure date dans z

x.annee=1992 ;     affecte la valeur 1992 au champ annee de la variable structurée x.

### 4. Structures imbriquées

Un champ d'une déclaration de structure peut être à nouveau une structure (attention : la récursivité est interdite : on verra ultérieurement pourquoi).

Exemple

```
struct identite{  
    char nom[16] ;  
    char prenom[20] ;  
    struct adr adresse ;  
    char No_secu[13] ;  
    struct date naissance;  
} individu ; (la variable individu est de type identite)
```

Avec struct date telle que définie déjà et

```
struct adr
{
    int numero ;
    char rue[40] ;
    int code_postal ;
    char localite[16] ;
};
```

Pour accéder à l'année de naissance de l'individu on utilise le chemin d'accès suivant :

individu.naissance.annee

### Exercice

Soit la structure suivante définissant un point

```
struct Point
{
    char Nom ;
    float x ;
    float y ;
};
```

Ecrire un programme qui donne des valeurs initiales à un point (on affichera ces valeurs initiales) puis qui demande à l'utilisateur de rentrer de nouvelles valeurs dans les coordonnées de ce point, et enfin qui affiche les champs de ce point.

## II Les pointeurs

### 1. Définition

Une variable pointeur est une variable contenant l'adresse d'une autre variable

En C, plusieurs opérateurs sont associés aux pointeurs :

- l'opérateur d'indirection \* (= « contenu de ») permet d'accéder à la valeur de l'objet sur lequel pointe le pointeur
- l'opérateur & (= « adresse de ») donne l'adresse d'une variable

### Exemple

Si px a été déclaré comme variable pointeur pointant sur un entier

Si x et y désignent des entiers

```
px=&x ; // px ← « adresse de x »
```

`y=*px ; // y ← « contenu de la variable dont l'adresse est dans px »`

ces deux lignes équivalent à

`y=x ;`

## 2. Déclaration de pointeur

Déclarer un pointeur consiste à :

- Déclarer la variable pointeur ;
- Déclarer le type des objets pointés par le pointeur

Syntaxe :

type de l'objet pointé \*nompointeur ;

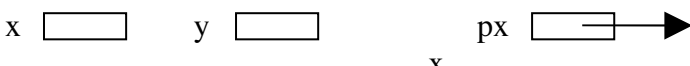
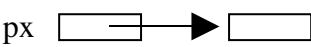
exemple :

`int *px //indique que px est un pointeur et que l'objet pointé est de type entier`

## 3. Utilisation des pointeurs

-Manipulation de l'objet pointé

Exemple

`int x,*px,y ;` schéma :   
`px=&x ;`  
`*px=0 ; // affecte la valeur 0 à x`  


`y>(*px)+1 ; //équivalent à y=x+1`

-Calcul d'adresse

Si p est un pointeur

`p=&y ; //affecte à p l'adresse de y`

`p=p+1 ; //p pointe maintenant sur l'objet suivant y`

L'incréméntation effective de l'adresse dépend du type de l'objet pointé (char =1octet, int=2 octets, etc)

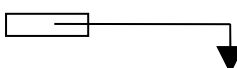
-notion de pointeur NULL

Un pointeur peut ne pointer sur aucune variable. Dans ce cas on écrit

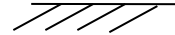
`int *pa ;`

`pa=0 ; // ou encore`

`pa=NULL ;`

schéma : 

#### 4. Pointeurs et fonctions



En C, les transmissions de paramètres aux fonctions se font par valeur (voir chapitre 2). Mais un paramètre formel d'une fonction peut être un pointeur, auquel cas la valeur du paramètre effectif correspondant sera en fait l'adresse d'une variable. Cette variable (dont l'adresse est contenue dans le pointeur) sera alors réellement modifiée par la fonction.

Prenons un exemple on cherche à permuter deux variables en utilisant une fonction. On écrit deux versions différentes de cette fonction : une sans pointeur, l'autre avec pointeur.

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

```
void echange1(int x, int y)
```

```
{
    int temp ;
    temp=x ;
    x=y ;
    y=temp ;
}
```

```
void echange2(int *px, int *py)
```

```
{
    int temp ;
    temp= *px ;
    *px= *py ;
    *py=temp ;
}
```

```
int main (void)
```

```
{
    int a,b ;
    printf(« entrer a et b\n » ;
    scanf(« %d %d », &a, &b) ;
    echange1(a,b) ;
    echange2(&a, &b) ;
    printf(« a=%d, b=%d », a,b) ;
}
```

Lors de l'appel de la fonction `echange1(a,b)` les valeurs de `a` et `b` ne seront pas modifiées donc `a` et `b` ne vont pas permuter. Par contre l'appel `echange2(&a,&b)` permettra de d'échanger les valeurs de `a` et de `b`.

Ainsi chaque fois que l'on veut modifier dans une fonction, une variable qui est un paramètre de cette fonction, alors le paramètre formel correspondant à cette variable doit être un pointeur vers un objet du même type.

Rappel et exemple : `scanf(« %d », &a) ;` => le `&` indique qu'il s'agit de l'adresse de la variable et permettra la modification de cette variable

### 5. Pointeur et tableaux

En C les tableaux sont considérés comme des constantes de type pointeur.

Le nom d'un tableau équivaut à l'adresse de son premier élément.

Ainsi si on a :

```
int a[10], *pa ;  
alors  
a équivaut à &a[0]
```

Ainsi

`pa = a` est équivalent à `pa = &a[0]`

De plus si on écrit :

```
pa = a ;  
alors (pa+1) est l'adresse de l'élément suivant a[1]  
*(a+1) } équivaut à a[1]  
*(pa+1) }
```

```
*(a+i) } équivaut à a[i]  
*(pa+i) }
```

Et

```
pa+1 } équivaut à &a[1]  
a+1 }
```

Remarque :

Un nom de tableau n'est pas un équivalent à une variable pointeur:

- une variable de type pointeur est une variable contenant une adresse
- un nom de tableau est une constante désignant une adresse

```
pa = a;
```

```
pa=pa+1;  
pa=pa-2;          ont un sens
```

mais

```
a=pa;  
a=a+1;  
pa=&a;            n'ont aucun sens car a est une constante de type pointeur
```

## 6. Tableaux de pointeurs

Il est possible de disposer de tableaux de pointeurs.

Exemple:

```
char *point[100];
```

Il s'agit de 100 pointeurs pointant chacun vers un caractère.

```
char t[50];  
char m[20];
```

```
point[0]=t;  
point[1]=m;  
alors  
point[0] pointe sur t[0]  
*point[0] est t[0]  
*(point[0]+1) est sur t[1], etc.  
point[1] pointe sur m[0]
```

Exercice

Comment écrire en C :

- a) un entier
- b) un pointeur sur un entier
- c) un tableau de 4 entiers
- d) un tableau de 4 pointeurs sur entier
- e) un pointeur sur un tableau de 4 entiers
- f) un pointeur sur un tableau de 4 pointeurs sur entier

## 7. Gestion de la taille d'un tableau

Jusqu'à présent, quand on déclarait un tableau, il fallait donner sa taille (plus exactement son nombre d'éléments). Mais cela nécessite de connaître la taille avant de d'écrire le programme.

Or ceci n'est pas toujours possible (par exemple, si l'on fait un programme qui doit calculer la moyenne d'une classe, on ne connaît pas à l'avance le nombre d'élèves, qui peut varier d'une classe à l'autre). Une solution consiste à réserver une place supérieure au maximum que l'on pourra rencontrer. Si cela est possible dans le cas précédent (par exemple, on peut prendre une taille de 100, si l'on considère qu'une classe a toujours moins de 100 élèves), ce n'est pas toujours le cas. Par exemple, si l'on fait un programme qui gère une bibliothèque, de nouveaux ouvrages arrivent tous les jours, et aucun maximum n'est envisageable. Ou alors le maximum que l'on peut considérer fera que l'on occupera toute la mémoire de l'ordinateur. Aussi il apparaît plus judicieux de réserver la place qu'une fois le besoin est connu (à l'exécution). Ceci est possible grâce à la fonction *malloc*, dont l'utilisation nécessite l'inclusion du fichier *stdlib.h*. Ces fonctions essaient de réserver en mémoire la place demandée. Si cela est possible, elles renvoient un pointeur sur l'emplacement réservé. Sinon elles renvoient le pointeur NULL (zéro).

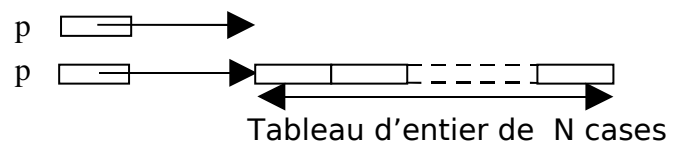
Exemple:

```
int *p; int N;
scanf(« %d », &N)
```

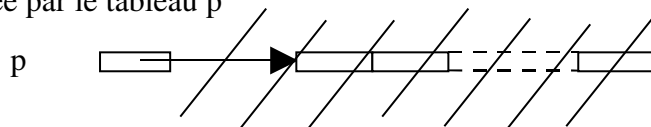
int T[N]; => impossible !!!

La solution:

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



free(p); => on libère la mémoire occupée par le tableau p



## 8. Opérations sur les pointeurs

Les opérations suivantes sur les pointeurs sont permises:

pointeur + entier Le résultat est un pointeur

pointeur - entier

pointeur1 - pointeur2

Les comparaisons entre pointeurs sont les suivantes:

== , !=, <, <=, >, >=

## III Structures et Pointeurs



## 1. Pointeurs de structures

Les seules opérations sont (pour l'instant):

- accès à un champ
- récupérer l'adresse de la structure (avec &)

Une structure peut être aussi recopiée ou peut être le résultat d'une fonction.

Reprenons la structure date vue précédemment :

```
struct date {  int jour ;
               int mois ;
               int annee ;
               char nom_mois[9] ;
               } ;
```

On peut déclarer des variables de type structure mais aussi des pointeurs pointant sur des structures.

```
struct date x,y, *pt ; //pt est un pointeur pointant vers une structure de type date
```

Pour accéder par exemple au champ de la variable pointé par pt on écrit :

```
(*pt).jour
```

On peut aussi écrire :

```
pt->jour //se lit « pt flèche jour »
```

## 2. Structures imbriquées et pointeurs

Il arrive parfois que le champ d'une structure donné ait exactement la même structure donnée.

On serait tenté d'écrire par exemple :

```
struct pers {
    char nom[20] ;
    char prenom[20] ;
    struct pers pere ;
    struct pers mere ;
};
```

```
struct pers moi;
```

Il se trouve que cette déclaration de structure pose problème. En effet, pour un tel programme, il faudrait réserver pour la variable moi la place nécessaire pour le nom, le prénom, le père et

la mère. La taille des tableaux nom et prénom sont connues mais que vaut celle du champ père ? Comme c'est une structure de type pers, il lui faut donc la place pour le nom, le prénom, son père (le grandpère de la variable moi) sa mère, etc. Donc il faudrait réserver une place mémoire infinie. Pour s'en sortir, on utilise des pointeurs :

```
struct pers {  
    char nom[20] ;  
    char prenom[20] ;  
    struct pers * pere ;  
    struct pers* mere ;  
};
```

```
struct pers moi;
```

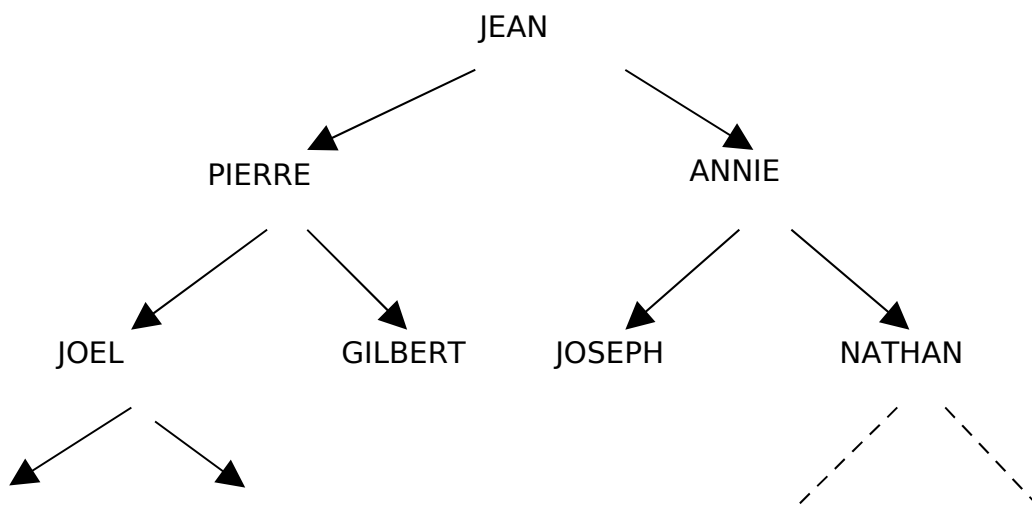
Cette version est correcte car lorsque l'on déclare une variable de type pointeur sur une structure, seule la place nécessaire au pointeur est réservée (et non celle de la structure). Et cette place est fixe.

L'utilisation des pointeurs sur les structures vont par exemple permettre d'utiliser des variables de type structure comme paramètres modifiables d'une fonction.

L'utilisation de pointeurs sur les structures permet aussi la représentation d'arbres binaires (arbres à deux branches) qu'on appelle « graphe ».

```
struct nœud {  
    char nom[20] ;  
    struct nœud *ss_arbre_gauche ; //pointeur vers un sous-arbre gauche  
    struct nœud *ss_arbre_droit ; //pointeur vers un sous-arbre droit  
};
```

```
struct JEAN ;
```



LUCIENNE      TONY

Nous donnerons uniquement quelques éléments de terminologie sur les arbres. Les éléments d'un arbre sont appelés des nœuds. Les nœuds possèdent une relation d'ordre ou de hiérarchie (que l'on assimile à la filiation). On dit qu'un nœud possède un père : c'est le nœud qui lui est supérieur dans la hiérarchie. Un nœud peut aussi avoir des fils.

Exemple

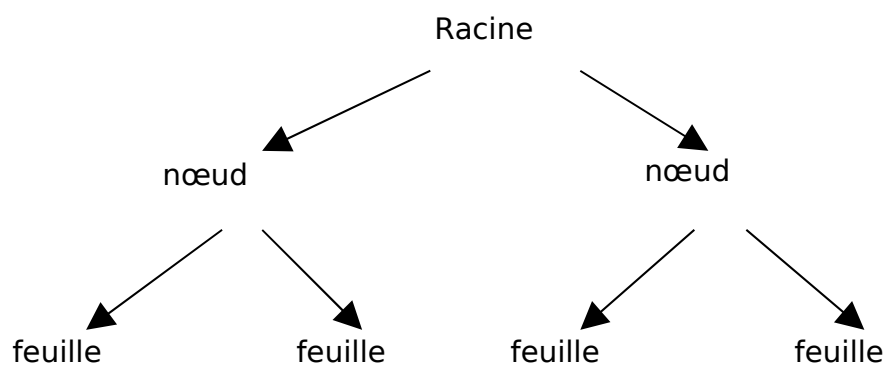
PIERRE a deux fils : JOEL et GILBERT. Le père de PIERRE est JEAN.

Il existe un nœud qui n'a pas de père : on l'appelle la racine.

Exemple : JEAN est la racine

Les nœuds qui n'ont pas de fils s'appellent les feuilles.

Exemple : LUCIENNE et TONY sont des feuilles



Un arbre où les nœuds ont au plus deux fils est appelé arbres binaire.