# Chapitre 3

# 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.

## 3.1 Adresse et valeur d'un objet

On appelle *Lvalue* (left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une *Lvalue* est caractérisée par :

– son adresse, c'est-à-dire l'adresse-mémoire à partir de laquelle l'objet est stocké ;

– sa valeur, c'est-à-dire ce qui est stocké à cette adresse.

Dans l'exemple,

```
int i, j;
i = 3;
j = i;
```

Si le compilateur a placé la variable `i` à l'adresse 4831836000 en mémoire, et la variable `j` à l'adresse 4831836004, on a

| objet | adresse | valeur |
|---|---|---|
| `i` | 4831836000 | 3 |
| `j` | 4831836004 | 3 |

Deux variables différentes ont des adresses différentes. L'affectation `i = j;` n'opère que sur les valeurs des variables. Les variables `i` et `j` étant de type `int`, elles sont stockées sur 4 octets. Ainsi la valeur de `i` est stockée sur les octets d'adresse 4831836000 à 4831836003.

L'adresse d'un objet étant un numéro d'octet en mémoire, il s'agit d'un entier quelque soit le type de l'objet considéré. Le format interne de cet entier (16 bits, 32 bits ou 64 bits) dépend des architectures. Sur un DEC alpha, par exemple, une adresse a toujours le format d'un entier long (64 bits).

L'opérateur `&` permet d'accéder à l'adresse d'une variable. Toutefois `&i` n'est pas une Lvalue mais une constante : on ne peut pas faire figurer `&i` à gauche d'un opérateur d'affectation. Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, les pointeurs.

## 3.2 Notion de pointeur

Un *pointeur* est un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction :

*type *nom-du-pointeur*;

où *type* est le type de l'objet pointé. Cette déclaration déclare un identificateur, *nom-du-pointeur*, associé à un objet dont la valeur est l'adresse d'un autre objet de type *type*. L'identificateur *nom-du-pointeur* est donc en quelque sorte un identificateur d'adresse. Comme pour n'importe quelle Lvalue, sa valeur est modifiable.

Même si la valeur d'un pointeur est toujours un entier (éventuellement un entier long), le type d'un pointeur dépend du type de l'objet vers lequel il pointe. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur. En effet, pour un pointeur sur un objet de type `char`, la valeur donne l'adresse de l'octet où cet objet est stocké. Par contre, pour un pointeur sur un objet de type `int`, la valeur donne l'adresse du premier des 4 octets où l'objet est stocké. Dans l'exemple suivant, on définit un pointeur `p` qui pointe vers un entier `i` :

```
int i = 3;
int *p;

p = &i;
```

On se trouve dans la configuration

| objet | adresse | valeur |
|---|---|---|
| i | 4831836000 | 3 |
| p | 4831836004 | 4831836000 |

L'*opérateur unaire d'indirection* `*` permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si `p` est un pointeur vers un entier `i`, `*p` désigne la valeur de `i`. Par exemple, le programme

```
main()
{
  int i = 3;
  int *p;

  p = &i;
  printf("*p = %d \n",*p);
}
```

imprime `*p = 3`.

Dans ce programme, les objets `i` et `*p` sont identiques : ils ont mêmes adresse et valeur. Nous sommes dans la configuration :

| objet | adresse | valeur |
|---|---|---|
| `i` | 4831836000 | 3 |
| `p` | 4831836004 | 4831836000 |
| `*p` | 4831836000 | 3 |

Cela signifie en particulier que toute modification de `*p` modifie `i`. Ainsi, si l'on ajoute l'instruction `*p = 0;` à la fin du programme précédent, la valeur de `i` devient nulle.

On peut donc dans un programme manipuler à la fois les objets `p` et `*p`. Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants :

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;
}
```

et

```
main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  p1 = p2;
}
```

Avant la dernière affectation de chacun de ces programmes, on est dans une configuration du type :

| objet | adresse | valeur |
|---|---|---|
| `i` | 4831836000 | 3 |
| `j` | 4831836004 | 6 |
| `p1` | 4831835984 | 4831836000 |
| `p2` | 4831835992 | 4831836004 |

Après l'affectation `*p1 = *p2;` du premier programme, on a

| objet | adresse | valeur |
|---|---|---|
| `i` | 4831836000 | 6 |
| `j` | 4831836004 | 6 |
| `p1` | 4831835984 | 4831836000 |
| `p2` | 4831835992 | 4831836004 |

Par contre, l'affectation `p1 = p2` du second programme, conduit à la situation :

| objet | adresse | valeur |
|---|---|---|
| `i` | 4831836000 | 3 |
| `j` | 4831836004 | 6 |
| `p1` | 4831835984 | 4831836004 |
| `p2` | 4831835992 | 4831836004 |

## 3.3 Arithmétique des pointeurs

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

- l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

Notons que la somme de deux pointeurs n'est pas autorisée.

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 * sizeof(`***type***`)`. Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation `++` et `--`. Par exemple, le programme

```
main()
{
  int i = 3;
  int *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

affiche `p1 = 4831835984        p2 = 4831835988`.

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

```
main()
{
  double i = 3;
  double *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

affiche `p1 = 4831835984          p2 = 4831835992`.

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. Ainsi, le programme suivant imprime les éléments du tableau `tab` dans l'ordre croissant puis décroissant des indices.

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int *p;
  printf("\n ordre croissant:\n");
  for (p = &tab[0]; p <= &tab[N-1]; p++)
    printf(" %d \n",*p);
  printf("\n ordre decroissant:\n");
  for (p = &tab[N-1]; p >= &tab[0]; p--)
    printf(" %d \n",*p);
}
```

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)/sizeof(`***type***`)` .

## 3.4 Allocation dynamique

Avant de manipuler un pointeur, et notamment de lui appliquer l'opérateur d'indirection `*`, il faut l'initialiser. Sinon, par défaut, la valeur du pointeur est égale à une constante symbolique notée `NULL` définie dans `stdio.h`. En général, cette constante vaut 0. Le test `p == NULL` permet de savoir si le pointeur `p` pointe vers un objet.

On peut initialiser un pointeur `p` par une affectation sur `p`. Par exemple, on peut affecter à `p` l'adresse d'une autre variable. Il est également possible d'affecter directement une valeur à `*p`. Mais pour cela, il faut d'abord réserver à `*p` un espace-mémoire de taille adéquate. L'adresse de cet espace-mémoire sera la valeur de `p`. Cette opération consistant à réserver un espace-mémoire pour stocker l'objet pointé s'appelle *allocation dynamique*. Elle se fait en C par la fonction `malloc` de la librairie standard `stdlib.h`. Sa syntaxe est

`malloc(`***nombre-octets***`)`

Cette fonction retourne un pointeur de type `char *` pointant vers un objet de taille ***nombre-octets*** octets. Pour initialiser des pointeurs vers des objets qui ne sont pas de type `char`, il faut convertir le type de la sortie de la fonction `malloc` à l'aide d'un cast. L'argument ***nombre-octets*** est souvent donné à l'aide de la fonction `sizeof()` qui renvoie le nombre d'octets utilisés pour stocker un objet.

Ainsi, pour initialiser un pointeur vers un entier, on écrit :

```
#include <stdlib.h>
int *p;
p = (int*)malloc(sizeof(int));
```

On aurait pu écrire également
`p = (int*)malloc(4);`
puisqu'un objet de type `int` est stocké sur 4 octets. Mais on préférera la première écriture qui a l'avantage d'être portable.

Le programme suivant

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int i = 3;
  int *p;
  printf("valeur de p avant initialisation = %ld\n",p);
  p = (int*)malloc(sizeof(int));
  printf("valeur de p apres initialisation = %ld\n",p);
  *p = i;
  printf("valeur de *p = %d\n",*p);
}
```

définit un pointeur `p` sur un objet `*p` de type `int`, et affecte à `*p` la valeur de la variable `i`. Il imprime à l'écran :

```
valeur de p avant initialisation = 0
valeur de p apres initialisation = 5368711424
valeur de *p = 3
```

Avant l'allocation dynamique, on se trouve dans la configuration

| objet | adresse | valeur |
|---|---|---|
| `i` | 4831836000 | 3 |
| `p` | 4831836004 | 0 |

A ce stade, `*p` n'a aucun sens. En particulier, toute manipulation de la variable `*p` générerait une violation mémoire, détectable à l'exécution par le message d'erreur `Segmentation fault`.

L'allocation dynamique a pour résultat d'attribuer une valeur à `p` et de réserver à cette adresse un espace-mémoire composé de 4 octets pour stocker la valeur de `*p`. On a alors

| objet | adresse | valeur |
|---|---|---|
| `i` | 4831836000 | 3 |
| `p` | 4831836004 | 5368711424 |
| `*p` | 5368711424 | ? (int) |

`*p` est maintenant définie mais sa valeur n'est pas initialisée. Cela signifie que `*p` peut valoir n'importe quel entier (celui qui se trouvait précédemment à cette adresse). L'affectation `*p = i;` a enfin pour résultat d'affecter à `*p` la valeur de `i`. A la fin du programme, on a donc

| objet | adresse | valeur |
|---|---|---|
| `i` | 4831836000 | 3 |
| `p` | 4831836004 | 5368711424 |
| `*p` | 5368711424 | 3 |

Il est important de comparer le programme précédent avec

```
main()
{
  int i = 3;
  int *p;

  p = &i;
}
```

qui correspond à la situation

| objet | adresse | valeur |
|---|---|---|
| i | 4831836000 | 3 |
| p | 4831836004 | 4831836000 |
| *p | 4831836000 | 3 |

Dans ce dernier cas, les variables `i` et `*p` sont identiques (elles ont la même adresse) ce qui implique que toute modification de l'une modifie l'autre. Ceci n'était pas vrai dans l'exemple précédent où `*p` et `i` avaient la même valeur mais des adresses différentes.

On remarquera que le dernier programme ne nécessite pas d'allocation dynamique puisque l'espace-mémoire à l'adresse `&i` est déjà réservé pour un entier.

La fonction `malloc` permet également d'allouer un espace pour plusieurs objets contigus en mémoire. On peut écrire par exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int i = 3;
  int j = 6;
  int *p;
  p = (int*)malloc(2 * sizeof(int));
  *p = i;
  *(p + 1) = j;
  printf("p = %ld \t *p = %d \t p+1 = %ld \t *(p+1) = %d \n",p,*p,p+1,*(p+1));
}
```

On a ainsi réservé, à l'adresse donnée par la valeur de `p`, 8 octets en mémoire, qui permettent de stocker 2 objets de type `int`. Le programme affiche
`p = 5368711424    *p = 3    p+1 = 5368711428    *(p+1) = 6` .

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. Sa syntaxe est

`calloc(`*nb-objets*`,`*taille-objets*`)`

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

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

est strictement équivalente à

```
p = (int*)malloc(N * sizeof(int));
for (i = 0; i < N; i++)
  *(p + i) = 0;
```

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

Enfin, lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement (c'est-à-dire quand on n'utilise plus le pointeur `p`), il faut libérer cette place en mémoire. Ceci se fait à l'aide de l'instruction `free` qui a pour syntaxe

`free(`*nom-du-pointeur*`);`

A toute instruction de type `malloc` ou `calloc` doit être associée une instruction de type `free`.

## 3.5 Pointeurs et tableaux

L'usage des pointeurs en C est, en grande partie, orienté vers la manipulation des tableaux.

### 3.5.1 Pointeurs et tableaux à une dimension

Tout tableau en C est en fait un pointeur constant. Dans la déclaration

```
int tab[10];
```

`tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, `tab` a pour valeur `&tab[0]`. On peut donc utiliser un pointeur initialisé à `tab` pour parcourir les éléments du tableau.

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    {
      printf(" %d \n",*p);
      p++;
    }
}
```

On accède à l'élément d'indice `i` du tableau `tab` grâce à l'opérateur d'indexation `[]`, par l'expression `tab[i]`. Cet opérateur d'indexation peut en fait s'appliquer à tout objet `p` de type pointeur. Il est lié à l'opérateur d'indirection `*` par la formule

`p[i]` = `*(p + i)`

Pointeurs et tableaux se manipulent donc exactement de même manière. Par exemple, le programme précédent peut aussi s'écrire

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    printf(" %d \n", p[i]);
}
```

Toutefois, la manipulation de tableaux, et non de pointeurs, possède certains inconvénients dûs au fait qu'un tableau est un pointeur constant. Ainsi

- on ne peut pas créer de tableaux dont la taille est une variable du programme,
- on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement. Ainsi, pour créer un tableau d'entiers à `n` éléments où `n` est une variable du programme, on écrit

```
#include <stdlib.h>
main()
{
  int n;
  int *tab;

   ...
  tab = (int*)malloc(n * sizeof(int));
   ...
  free(tab);
}
```

Si on veut en plus que tous les éléments du tableau `tab` soient initialisés à zéro, on remplace l'allocation dynamique avec `malloc` par

```
  tab = (int*)calloc(n, sizeof(int));
```

Les éléments de `tab` sont manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux.

Les deux différences principales entre un tableau et un pointeur sont

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple `p = &i` ;
- un tableau n'est pas une Lvalue ; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire `tab++;`).

### 3.5.2 Pointeurs et tableaux à plusieurs dimensions

Un tableau à deux dimensions est, par définition, un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur. Considérons le tableau à deux dimensions défini par :

```
int tab[M][N];
```

`tab` est un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier. `tab` a une valeur constante égale à l'adresse du premier élément du tableau, `&tab[0][0]`. De même `tab[i]`, pour `i` entre 0 et `M-1`, est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice `i`. `tab[i]` a donc une valeur constante qui est égale à `&tab[i][0]`.

Exactement comme pour les tableaux à une dimension, les pointeurs de pointeurs ont de nombreux avantages sur les tableaux multi-dimensionnés.

On déclare un pointeur qui pointe sur un objet de type *type* * (deux dimensions) de la même manière qu'un pointeur, c'est-à-dire

*type* ***nom-du-pointeur*;

De même un pointeur qui pointe sur un objet de type *type* ** (équivalent à un tableau à 3 dimensions) se déclare par

*type* ****nom-du-pointeur*;

Par exemple, pour créer avec un pointeur de pointeur une matrice à `k` lignes et `n` colonnes à coefficients entiers, on écrit :

```
main()
{
  int k, n;
  int **tab;

  tab = (int**)malloc(k * sizeof(int*));
  for (i = 0; i < k; i++)
    tab[i] = (int*)malloc(n * sizeof(int));
      ....

  for (i = 0; i < k; i++)
    free(tab[i]);
  free(tab);
}
```

La première allocation dynamique réserve pour l'objet pointé par `tab` l'espace-mémoire correspondant à `k` pointeurs sur des entiers. Ces `k` pointeurs correspondent aux lignes de la matrice. Les allocations dynamiques suivantes réservent pour chaque pointeur `tab[i]` l'espace-mémoire nécessaire pour stocker `n` entiers.

Si on désire en plus que tous les éléments du tableau soient initialisés à zéro, il suffit de remplacer l'allocation dynamique dans la boucle `for` par

```
    tab[i] = (int*)calloc(n, sizeof(int));
```

Contrairement aux tableaux à deux dimensions, on peut choisir des tailles différentes pour chacune des lignes `tab[i]`. Par exemple, si l'on veut que `tab[i]` contienne exactement `i+1` éléments, on écrit

```
for (i = 0; i < k; i++)
  tab[i] = (int*)malloc((i + 1) * sizeof(int));
```

### 3.5.3 Pointeurs et chaînes de caractères

On a vu précédemment qu'une chaîne de caractères était un tableau à une dimension d'objets de type `char`, se terminant par le caractère nul `'\0'`. On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`. On peut faire subir à une chaîne définie par

```
char *chaine;
```

des affectations comme

```
chaine = "ceci est une chaine";
```

et toute opération valide sur les pointeurs, comme l'instruction `chaine++;`. Ainsi, le programme suivant imprime le nombre de caractères d'une chaîne (sans compter le caractère nul).

```
#include <stdio.h>
main()
{
  int i;
  char *chaine;

  chaine = "chaine de caracteres";
  for (i = 0; *chaine != '\0'; i++)
    chaine++;
  printf("nombre de caracteres = %d\n",i);
}
```

La fonction donnant la longueur d'une chaîne de caractères, définie dans la librairie standard `string.h`, procède de manière identique. Il s'agit de la fonction `strlen` dont la syntaxe est

`strlen(`*chaine*`);`

où *chaine* est un pointeur sur un objet de type `char`. Cette fonction renvoie un entier dont la valeur est égale à la longueur de la chaîne passée en argument (moins le caractère `'\0'`). L'utilisation de pointeurs de caractère et non de tableaux permet par exemple de créer une chaîne correspondant à la concaténation de deux chaînes de caractères :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
```

```
  int i;
  char *chaine1, *chaine2, *res, *p;

  chaine1 = "chaine ";
  chaine2 = "de caracteres";
  res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
  p = res;
  for (i = 0; i < strlen(chaine1); i++)
    *p++ = chaine1[i];
  for (i = 0; i < strlen(chaine2); i++)
    *p++ = chaine2[i];
  printf("%s\n",res);
}
```

On remarquera l'utilisation d'un pointeur intermédiaire `p` qui est indispensable dès que l'on fait des opérations de type incrémentation. En effet, si on avait incrémenté directement la valeur de `res`, on aurait évidemment "perdu" la référence sur le premier caractère de la chaîne. Par exemple,

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  int i;
  char *chaine1, *chaine2, *res;

  chaine1 = "chaine ";
  chaine2 = "de caracteres";
  res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
  for (i = 0; i < strlen(chaine1); i++)
    *res++ = chaine1[i];
  for (i = 0; i < strlen(chaine2); i++)
    *res++ = chaine2[i];
  printf("\nnombre de caracteres de res = %d\n",strlen(res));
}
```

imprime la valeur `0`, puisque `res` a été modifié au cours du programme et pointe maintenant sur le caractère nul.

## 3.6 Pointeurs et structures

### 3.6.1 Pointeur sur une structure

Contrairement aux tableaux, les objets de type structure en C sont des Lvalues. Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier membre de la structure. On peut donc manipuler des pointeurs sur des structures. Ainsi, 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`*

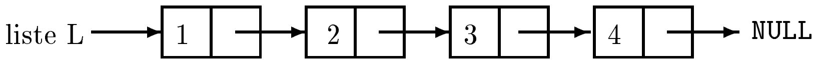
L'usage de parenthèses est ici indispensable car l'opérateur d'indirection `*` à une priorité plus élevée que l'opérateur de membre de structure. Cette notation peut être simplifiée grâce à l'opérateur *pointeur de membre de structure*, noté `->`. L'expression précédente est strictement équivalente à

`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`.

### 3.6.2 Structures auto-référencées

On a souvent besoin en C de modèles de structure dont un des membres est un pointeur vers une structure de même modèle. Cette représentation permet en particulier de construire des listes chaînées. En effet, il est possible de représenter une liste d'éléments de même type par un tableau (ou un pointeur). Toutefois, cette représentation, dite *contiguë*, impose que la taille maximale de la liste soit connue a priori (on a besoin du nombre d'éléments du tableau lors de l'allocation dynamique). Pour résoudre ce problème, on utilise une représentation *chaînée* : l'élément de base de la chaîne est une structure appelée *cellule* qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur la liste vide `NULL`. La liste est alors définie comme un pointeur sur le premier élément de la chaîne.




Pour représenter une liste d'entiers sous forme chaînée, on crée le modèle de structure `cellule` qui a deux champs : un champ `valeur` de type `int`, et un champ `suivant` de type pointeur sur une `struct cellule`. Une liste sera alors un objet de type pointeur sur une `struct cellule`. Grâce au mot-clef `typedef`, on peut définir le type `liste`, synonyme du type pointeur sur une `struct cellule`.

```
struct cellule
{
  int valeur;
  struct cellule *suivant;
};

typedef struct cellule *liste;
```

Un des avantages de la représentation chaînée est qu'il est très facile d'insérer un élément à un endroit quelconque de la liste. Ainsi, pour insérer un élément en tête de liste, on utilise la fonction suivante :

```
liste insere(int element, liste Q)
{
  liste L;
  L = (liste)malloc(sizeof(struct cellule));
  L->valeur = element;
  L->suivant = Q;
  return(L);
}
```

Le programme suivant crée une liste d'entiers et l'imprime à l'écran :

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

struct cellule
{
  int valeur;
```

```
  struct cellule *suivant;
};

typedef struct cellule *liste;

liste insere(int element, liste Q)
{
  liste L;
  L = (liste)malloc(sizeof(struct cellule));
  L->valeur = element;
  L->suivant = Q;
  return(L);
}

main()
{
  liste L, P;

  L = insere(1,insere(2,insere(3,insere(4,NULL))));
  printf("\n impression de la liste:\n");
  P = L;
  while (P != NULL)
    {
      printf("%d \t",P->valeur);
      P = P->suivant;
    }
}
```

On utilisera également une structure auto-référencée pour créer un arbre binaire :

```
struct noeud
{
  int valeur;
  struct noeud *fils_gauche;
  struct noeud *fils_droit;
};

typedef struct noeud *arbre;
```