Chapitre I

Les pointeurs

1. L'importance des pointeurs

- On peut accéder aux données en mémoire à l'aide de pointeurs i.e. des variables pouvant contenir des adresses d'autres variables.
- Comme nous le verrons dans le chapitre suivant, en C, les pointeurs jouent un rôle primordial dans la définition de fonctions :

  Les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions.

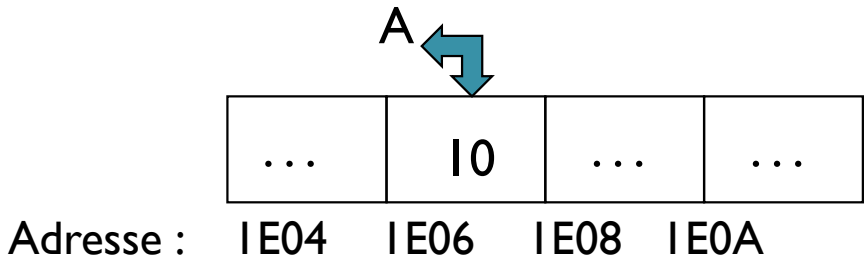
- Le traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs.
- Les pointeurs nous permettent de définir de nouveaux types de données : les piles, les files, les listes, ...
- Les pointeurs nous permettent d'écrire des programmes plus compacts et plus efficaces.
- Mais si l'on n'y prend pas garde, les pointeurs sont une excellente technique permettant de formuler des programmes incompréhensibles.

2. Mode d'adressage direct des variables

Adressage direct :

- Jusqu'à maintenant, nous avons surtout utilisé des variables pour stocker des informations.
- La valeur d'une variable se trouve à un endroit spécifique dans la mémoire de l'ordinateur.

```
short A;
A = 10;
```



- Le nom de la variable nous permet alors d'accéder directement à cette valeur.

Dans l'adressage direct, l'accès au contenu d'une variable se fait via le nom de la variable.

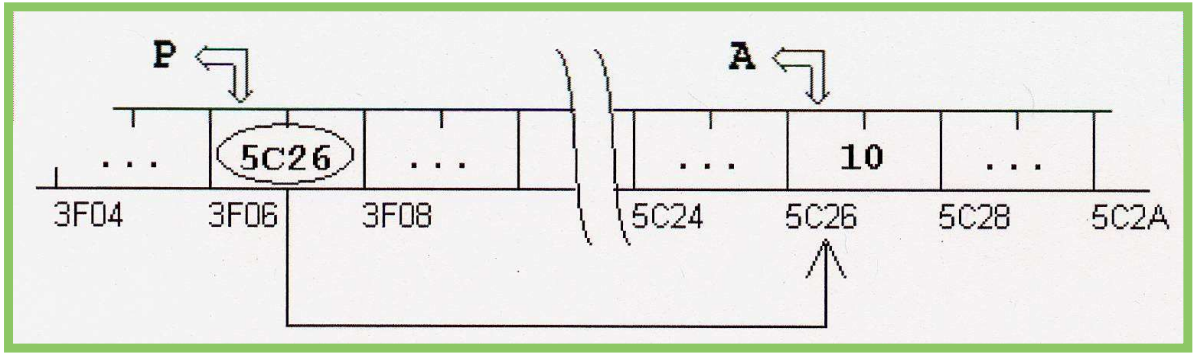
1. Mode d'adressage indirect des variables

Adressage indirect :

- Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, nous pouvons copier l'adresse de cette variable dans une variable spéciale, disons P, appelée pointeur.
- Nous pouvons alors retrouver l'information de la variable A en passant par le pointeur P.

Dans l'adressage indirect, l'accès au contenu d'une variable se fait via un pointeur qui renferme l'adresse de la variable.

Exemple : Soit A une variable renfermant la valeur 10, et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit :



3. Définition d'un pointeur

Un pointeur est une variable spéciale pouvant contenir l'adresse d'une autre variable.

- En C, chaque pointeur est limité à un type de données. Il ne peut contenir que l'adresse d'une variable de ce type. Cela élimine plusieurs sources d'erreurs.

Syntaxe permettant de déclarer un pointeur :

type de donnée * identificateur de variable pointeur;

Ex. : int * pNombre;

pNombre désigne une variable pointeur pouvant contenir uniquement l'adresse d'une variable de type int.

Si pNombre contient l'adresse d'une variable entière A, on dira alors que pNombre pointe vers A.

- Les pointeurs et les noms de variables ont le même rôle : ils donnent accès à un emplacement en mémoire.
  Par contre, un pointeur peut contenir différentes adresses mais le nom d'une variable (pointeur ou non) reste toujours lié à la même adresse.
- Bonne pratique de programmation : choisir des noms de variable appropriés (Ex. : pNombre, NombrePtr).

3.1 Comment obtenir l'adresse d'une variable ?

- Pour obtenir l'adresse d'une variable, on utilise l'opérateur & précédant le nom de la variable.

Syntaxe permettant d'obtenir l'adresse d'une variable :

& nom de la variable

Ex. :
```
int A;
int * pNombre = &A;
```

ou encore,

```
int A;
int * pNombre;
pNombre = &A;
```

pNombre désigne une variable pointeur initialisée à l'adresse de la variable A de type int.

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

scanf a besoin de l'adresse de chaque paramètre pour pouvoir lui attribuer une nouvelle valeur.

Note : L'opérateur & ne peut pas être appliqué à des constantes ou des expressions.

3.2 Comment accéder au contenu d'une adresse ?

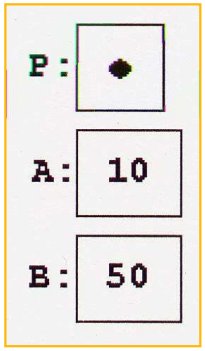
- Pour avoir accès au contenu d'une adresse, on utilise l'opérateur * précédant le nom du pointeur.

Syntaxe permettant d'avoir accès au contenu d'une adresse :

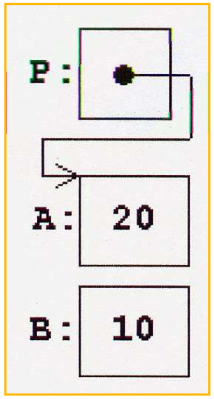
```
* nom du pointeur
```

Ex. :

```
int A = 10, B = 50;
int * P;
```




```
P = &A;
B = *P;
*P = 20;
```



*P et A désigne le même emplacement mémoire et *P peut être utilisé partout où on peut écrire A.

3.3 Priorité des opérateurs * et &

- Ces 2 opérateurs ont la même priorité que les autres opérateurs unaires (!, ++, --).

- Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

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++
```

Parenthèses obligatoires sans quoi, cela donne lieu à un accès non autorisé.

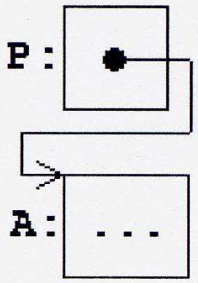
3.4 Le pointeur NULL

- Pour indiquer qu'un pointeur pointe nulle part, on utilise l'identificateur NULL (On doit inclure stdio.h ou iostream.h).
- On peut aussi utiliser la valeur numérique 0 (zéro).

```
int * P = 0;

if (P == NULL) printf("P pointe nulle part");
```

En résumé



Après les instructions:

```
int A;
int *P;
P = &A;
```

`A` désigne le contenu de A

`&A` désigne l'adresse de A

`P` désigne l'adresse de A

`*P` désigne le contenu de A

En outre:

`&P` désigne l'adresse du pointeur P

`*A` est illégal (puisque A n'est pas un pointeur)

A == *P ⇔ P == &A

A == *&A et P == &*P

4. Pointeurs et tableaux

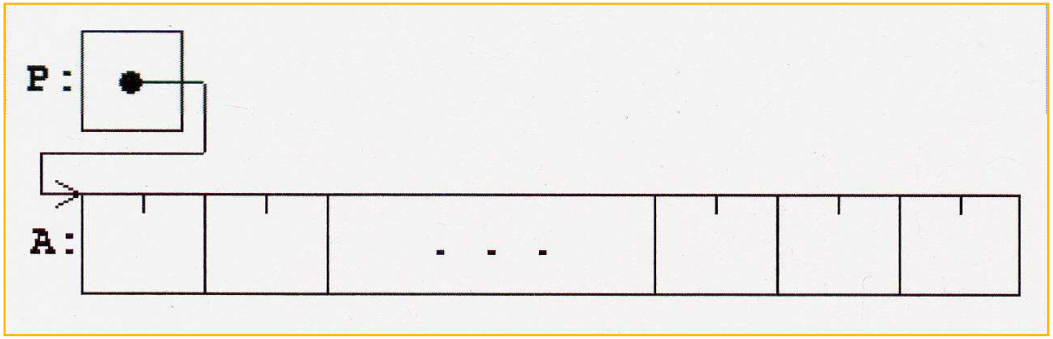
- Chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs.

- Comme nous l'avons déjà constaté, le nom d'un tableau représente l'adresse de la première composante.

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

- Le nom d'un tableau est un pointeur **constant** sur le premier élément du tableau.

```
int A[10];
int * P;
P = A;      est équivalente à     P = &A[0];
```



4.1 Adressage des composantes d'un tableau

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

  P + i pointe sur la i<u>ième</u> composante à droite de *P.
  P - i pointe sur la i<u>ième</u> composante à gauche de *P.

Ainsi, après l'instruction P = A;

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

- Incrémentation et décrémentation d'un pointeur

Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction

`P++;`	P pointe sur A[i+1]
`P+=n;`	P pointe sur A[i+n]
`P--;`	P pointe sur A[i-1]
`P-=n;`	P pointe sur A[i-n]

Ces opérateurs (+, -, ++, --, +=, -=) sont définis seulement à l'intérieur d'un tableau car on en peut pas présumer que 2 variables de même type sont stockées de façon contiguë en mémoire.

4.2 Calcul d'adresse des composantes d'un tableau

Note : Il peut paraître surprenant que P + i n'adresse pas le $i^{ième}$ octet après P, mais la $i^{ième}$ composante après P.

Pourquoi ? Pour tenter d'éviter des erreurs dans le calcul d'adresses.

Comment ? Le calcul automatique de l'adresse P + i est possible car, chaque pointeur est limité à un seul type de données, et le compilateur connaît le nombre d'octets des différents types.

Soit A un tableau contenant des éléments du type **float** et P un pointeur sur **float**:

```
float A[20], X;
float *P;
```

Après les instructions,

```
P = A;
X = *(P+9);
```

X contient la valeur du dixième élément de A, celle de A[9].

Une donnée de type float ayant besoin de 4 octets, le compilateur obtient l'adresse P + 9 en ajoutant 9 * 4 = 36 octets à l'adresse dans P.

4.3 Soustraction et comparaison de 2 pointeurs

- Soustraction de deux pointeurs

Soient P1 et P2 deux pointeurs qui pointent *dans le même tableau*:
`P1-P2` fournit le nombre de composantes comprises entre P1 et P2.

Le résultat de la soustraction `P1-P2` est
- négatif, si P1 précède P2
- zéro, si P1 = P2
- positif, si P2 precède P1
- indéfini, si P1 et P2 ne pointent pas dans le même tableau

- Comparaison de deux pointeurs

On peut comparer deux pointeurs par `<`, `>`, `<=`, `>=`, `==`, `!=`.

Mêmes tableaux : Comparaison des indices correspondants.

Tableaux différents : Comparaison des positions relatives en mémoire.

4.4 Différence entre un pointeur et le nom d'un tableau

Comme A représente l'adresse de A[0],

`*(A+1)` désigne le contenu de A[1]

`*(A+2)` désigne le contenu de A[2]

...

`*(A+i)` désigne le contenu de A[i]

- Un *pointeur* est une variable,
donc des opérations comme `P = A` ou `P++` sont permises.

- Le *nom d'un tableau* est une constante,
donc des opérations comme `A = P` ou `A++` sont impossibles.

Exemples

```
int tab[5]={2,1,0,8,4};
int* p;
p=tab;//ou p=&tab[0];  adresse du premier élément
for(int i=0 ; i<5 ; i++)
{
  printf ("%d \n",*p);
  p++;
}
```

Qui est équivalent à :

```
int tab[5]={2,1,0,8,4};
int* p;
p=tab;
for(int i=0 ; i<5 ; i++)
  printf("%d\n",p[i]);//car *(p+i)=p[i]
```

Qui est équivalent à :

```
int tab[5]={2,1,0,8,4};
int* p;
for(p=&tab[0] ; p<&tab[N]; p++)
  printf("%d \n",*p);
```

Résumons

Soit un tableau A de type quelconque et i un indice d'une composante de A,

`A`	désigne l'adresse de	`A[0]`
`A+i`	désigne l'adresse de	`A[i]`
`*(A+i)`	désigne le contenu de	`A[i]`

Si P = A, alors

`P`	pointe sur l'élément	`A[0]`
`P+i`	pointe sur l'élément	`A[i]`
`*(P+i)`	désigne le contenu de	`A[i]`

Activité

Ecrire un programme C permettant de charger un tableau de 5 entiers et d'inverser les éléments du tableau.

Utiliser la notion de pointeur.

5. Tableaux de pointeurs

Syntaxe :

type * identificateur du tableau[nombre de composantes];

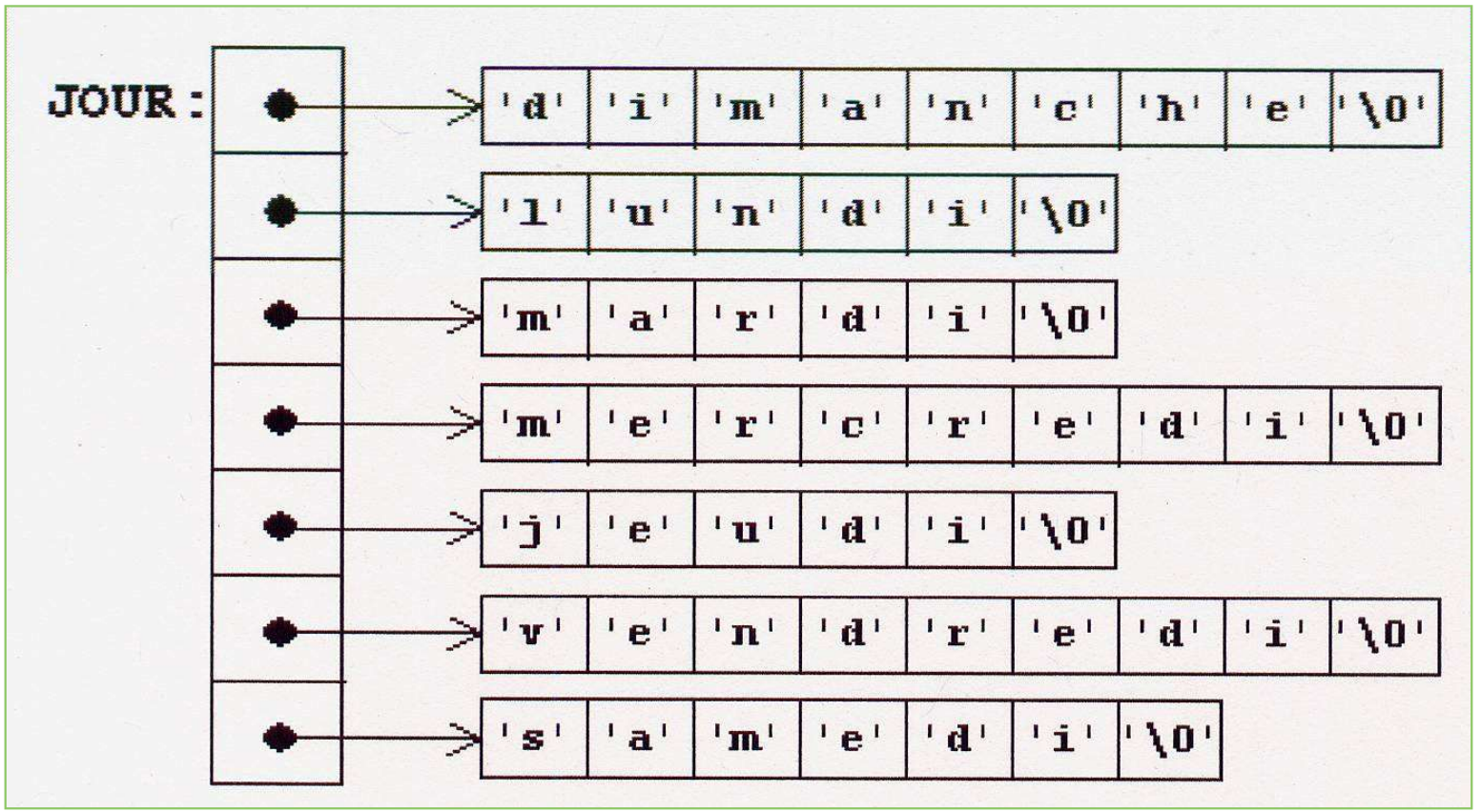
Exemple : `int * A[10];` // un tableau de 10 pointeurs
// vers des valeurs de type int.

Tableaux de pointeurs vers des chaînes de caractères de différentes longueurs

Exemple

```
char *JOUR[] = {"dimanche", "lundi", "mardi",
                "mercredi", "jeudi", "vendredi",
                "samedi"};
```

Nous avons déclaré un tableau JOUR[] de 7 pointeurs de type char, chacun étant initialisé avec l'adresse de l'une des 7 chaînes de caractères.



Affichage :

```
int I;
for (I=0; I<7; I++) printf("%s\n", JOUR[I]);
```

Pour afficher la 1e lettre de chaque jour de la semaine, on a :

```
int I;
for (I=0; I<7; I++) printf("%c\n", *JOUR[I]);
```

6. Allocation dynamique de la mémoire

Problématique :

Souvent, nous devons travailler avec des données dont nous ne pouvons prévoir le nombre et la grandeur lors de l'écriture du programme.

La taille des données est connue au temps d'exécution seulement.

Il faut éviter le gaspillage qui consiste à réserver l'espace maximal prévisible.

But : Nous cherchons un moyen de réserver ou de libérer de l'espace mémoire au fur et à mesure que nous en avons besoin pendant l'exécution du programme.

Exemples : La mémoire sera allouée au temps d'exécution.

```
char * P;          // P pointera vers une chaîne de caractères
                   // dont la longueur sera connue au temps d'exécution.

int * M[10];       // M permet de représenter une matrice de 10 lignes
                   // où le nombre de colonnes varie pour chaque ligne.
```

6.1 La fonction malloc et l'opérateur sizeof

- La fonction malloc de la bibliothèque stdlib nous aide à localiser et à réserver de la mémoire au cours de l'exécution d'un programme.

- La fonction malloc fournit l'adresse d'un bloc en mémoire disponible de N octets.

```
char * T = malloc(4000);
```

Cela fournit l'adresse d'un bloc de 4000 octets disponibles et l'affecte à T. S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

- Si nous voulons réserver de l'espace pour des données d'un type dont la grandeur varie d'une machine à l'autre, on peut se servir de sizeof pour connaître la grandeur effective afin de préserver la portabilité du programme.

```
sizeof <var>
   fournit la grandeur de la variable <var>
sizeof <const>
   fournit la grandeur de la constante <const>
sizeof (<type>)
   fournit la grandeur pour un objet du type <type>
```

Exemple :

```
#include <stdio.h>
void main()
{
        short A[10];
        char B[5][10];
        printf("%d%d%d%d%d%d", sizeof A, sizeof B,
                               sizeof 4.25,
                               sizeof "Bonjour !",
                               sizeof(float),
                               sizeof(double));
}
```

205081048

Exemple :

Réserver de la mémoire pour X valeurs de type int où X est lue au clavier.

```
int X;
int *PNum;
printf("Introduire le nombre de valeurs :");
scanf("%d", &X);
PNum = malloc(X*sizeof(int));
```

6.2 Libération de l'espace mémoire (free)

- Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de malloc, nous pouvons le libérer à l'aide de la fonction free de la librairie stdlib.

```
free(pointeur)
;
```

Pointe vers le bloc à libérer.

À éviter : Tenter de libérer de la mémoire avec free laquelle n'a pas été allouée par malloc.

Attention : La fonction free ne change pas le contenu du pointeur. Il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était rattaché.

Note : Si la mémoire n'est pas libérée explicitement à l'aide de free, alors elle l'est automatiquement à la fin de l'exécution du programme.