I) Adressage de variables :

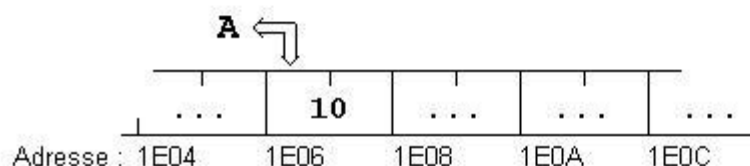
Avant de parler de pointeurs, il est indiqué de passer brièvement en revue les deux modes d'adressage principaux, qui vont d'ailleurs nous accompagner tout au long des chapitres suivants.

1) Adressage direct :

Dans la programmation, nous utilisons des variables pour stocker des informations. La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder *directement* à cette valeur.

Adressage direct : Accès au contenu d'une variable par le nom de la variable.

Exemple :



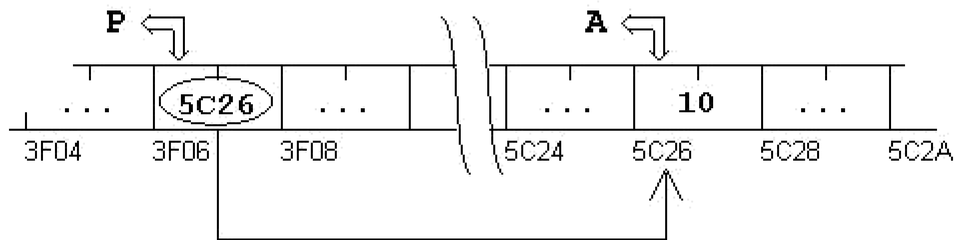
2) Adressage indirect :

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 P, appelée* ***pointeur.*** Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.

Adressage indirect : Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

Exemple :

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



II) Les pointeurs :

Définition : Pointeur

Un **pointeur** est une variable spéciale qui peut contenir l'**adresse** d'une autre variable.

En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.

Si un pointeur P contient l'adresse d'une variable A, on dit que

'P pointe sur A'.

Remarque :

Les pointeurs et les noms de variables ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :

* Un ***pointeur*** est une variable qui peut 'pointer' sur différentes adresses.
* Le ***nom d'une variable*** reste toujours lié à la même adresse.

1) Les opérateurs de base :

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.

a) 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'est-à-dire 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

P: •

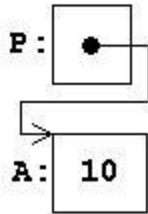
et A une variable (du même type) contenant la valeur 10 :

A: 10

Alors l'instruction

```
P=&A;
```

affecte l'adresse de la variable A à la variable P. Dans notre représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche :

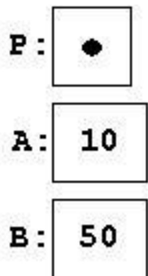


b) <u>L'opérateur 'contenu de' : *</u>

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

Exemple :

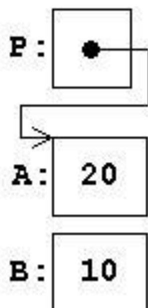
Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé :



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.



c) <u>Déclaration d'un pointeur :</u>

```
<Type> *<NomPointeur>
déclare un pointeur <NomPointeur> qui peut recevoir des
adresses de variables du type <Type>
```

Une déclaration comme

```
int *PNUM;
```

peut être interprétée comme suit :

*"*PNUM est du type **int**"*

ou

*"PNUM est un pointeur sur **int**"*

ou

*"PNUM peut contenir l'adresse d'une variable du type **int**"*

Exemple :

Le programme complet effectuant les transformations de l'exemple ci-dessus peut se présenter comme suit :

```
|main()                     ou bien   |main()
|{                                    |{
| /* déclarations */                  |/* déclarations */
| short A = 10;                       | short A, B, *P;
| short B = 50;                       | /* traitement */
| short *P;                           |A=10;
| /* traitement */                    |B=50;
|P=&A;                                |P=&A;
|B=*P;                                |B=*P;
|*P=20;                               |*P=20;
| return (0);                         | return (0);
|}                                    |}
```

Remarque :

Lors de la déclaration d'un pointeur en C, ce pointeur est lié explicitement à un type de données. Ainsi, la variable PNUM déclarée comme pointeur sur **int** ne peut pas recevoir l'adresse d'une variable d'un autre type que **int**.

Nous allons voir que la limitation d'un pointeur à un type de variables n'élimine pas seulement un grand nombre de sources d'erreurs très désagréables, mais permet une série d'opérations très pratiques sur les pointeurs.

2) Les opérations élémentaires sur pointeurs :

En travaillant avec des pointeurs, nous devons observer les règles suivantes :

a) Priorité de :

- * et &
- Les opérateurs ***** et **&** ont la même priorité que les autres opérateurs unaires (la négation **!**, l'incrémentation ++, la décrémentation **--**). Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.
- Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple :

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++
```

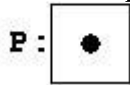
Dans le dernier cas, les parenthèses sont nécessaires :

Comme les opérateurs unaires * et ++ sont évalués *de droite à gauche*, sans les parenthèses le *pointeur* P serait incrémenté, *non pas l'objet* sur lequel P pointe.

On peut uniquement affecter des adresses à un pointeur.

b) Le pointeur NUL :

Seule exception : La valeur numérique 0 (zéro) est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.



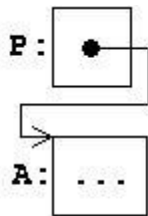

```
int *P;
P=0;
```

Finalement, les pointeurs sont aussi des variables et peuvent être utilisés comme telles. Soit P1 et P2 deux pointeurs sur **int**, alors l'affectation

```
P1 = P2;
```

copie le contenu de P2 vers P1. P1 pointe alors sur le même objet que P2.

c) Résumons :



Après les instructions :

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

A désigne le contenu de

&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)

HHH) Pointeurs et tableaux :

En C, il existe une relation très étroite entre tableaux et pointeurs. Ainsi, chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs. En général, les versions formulées avec des pointeurs sont plus compactes et plus efficientes, surtout à l'intérieur de fonctions. Mais, du moins pour des débutants, le 'formalisme pointeur' est un peu inhabituel.

1) Adressage des composantes d'un tableau :

Comme nous l'avons déjà constaté, le nom d'un tableau représente l'adresse de son premier élément. En d'autres 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.

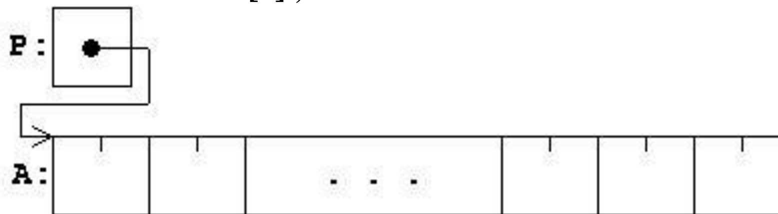
Exemple :

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]

Remarques :

Au premier coup d'œil, il est bien surprenant que P+i n'adresse pas le i-ième ***octet*** derrière P, mais la i-ième ***composante*** derrière P ...

Ceci s'explique par la stratégie de programmation 'défensive' des créateurs du langage C :

Si on travaille avec des pointeurs, les erreurs les plus perfides sont causées par des pointeurs mal placés et des adresses mal calculées. En C, le compilateur peut calculer automatiquement l'adresse de l'élément P+i en ajoutant à P la grandeur d'une composante multipliée par i. Ceci est possible, parce que :

- chaque pointeur est limité à un seul type de données, et
- le compilateur connaît le nombre d'octets des différents types.

Exemple :

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 10-ième élément de A, (c'est-à-dire celle de A[9]). Une donnée du 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.

Rassemblons les constatations ci dessus :

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]

Attention !

Il existe toujours une différence essentielle entre un pointeur et le nom d'un tableau :

- 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.

Ceci nous permet de jeter un petit coup d'œil derrière les rideaux :

Lors de la première phase de la compilation, toutes les expressions de la forme A[i] sont traduites en *(A+i). En multipliant l'indice i par la grandeur d'une composante, on obtient un indice en octets :

*<indice en octets> = <indice élément> * <grandeur élément>*

Cet indice est ajouté à l'adresse du premier élément du tableau pour obtenir l'adresse de la composante i du tableau. Pour le calcul d'une adresse donnée par une adresse plus un indice en octets, on utilise un mode d'adressage spécial connu sous le nom ***'adressage indexé'*** *:*

<adresse indexée> = <adresse> + <indice en octets>

Presque tous les processeurs disposent de plusieurs registres spéciaux *(registres index)* à l'aide desquels on peut effectuer l'adressage indexé de façon très efficace.

Résumons :

Soit un tableau A d'un type quelconque et i un indice pour les composantes de A, alors

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]**

Formalisme tableau et formalisme pointeur :

A l'aide de ce bagage, il nous est facile de 'traduire' un programme écrit à l'aide du *'formalisme tableau'* dans un programme employant le *'formalisme pointeur'*.

Exemple :

Les deux programmes suivants copient les éléments positifs d'un tableau T dans un deuxième tableau POS.

Formalisme tableau :

```
main()
{
 int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -
 9}; int POS[10];
 int I,J; /* indices courants dans T et POS */
 for (J=0,I=0 ; I<10 ; I++)
     if (T[I]>0)
         {
          POS[J] = T[I];
          J++;
         }
 return (0);
}
```

Nous pouvons remplacer systématiquement la notation **tableau[I]** par ***(tableau + I)**, ce qui conduit à ce programme :

Formalisme pointeur :

```
main()
```

```
{
 int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -
 9}; int POS[10];
 int I,J; /* indices courants dans T et POS */
 for (J=0,I=0 ; I<10 ; I++)
     if (*(T+I)>0)
         {
          *(POS+J) = *(T+I);
          J++;
         }
 return (0);
}
```

Sources d'erreurs :

Un bon nombre d'erreurs lors de l'utilisation de C provient de la confusion entre soit contenu et adresse, soit pointeur et variable. Revoyons donc les trois types de déclarations que nous connaissons jusqu'ici et résumons les possibilités d'accès aux données qui se présentent.

Les *variables et leur utilisation* `int A` ; déclare une ***variable simple*** du type `int`

A désigne *le contenu de A*

&A désigne l'adresse de A

int B[] ; déclare un ***tableau*** d'éléments du type **int**

B désigne *l'adresse de la première composante de B*.
(Cette adresse est toujours constante)

B[i] désigne le contenu de la composante i du tableau

&B[i] désigne l'adresse de la composante i du tableau

en utilisant le formalisme pointeur :

B+i désigne l'adresse de la composante i du tableau

***(B+i)** désigne le contenu de la composante i du tableau

int *P ; déclare un ***pointeur*** sur des éléments du type **int**.

P peut pointer sur des variables simples du type **int** ou
sur les composantes d'un tableau du type **int**.

2) Arithmétique des pointeurs :

Comme les pointeurs jouent un rôle si important, le langage C soutient une série d'opérations arithmétiques sur les pointeurs que l'on ne rencontre en général que dans les langages machines. Le confort de ces opérations en C est basé sur le principe suivant :

Toutes les opérations avec les pointeurs tiennent compte automatiquement du type et de la grandeur des objets pointés.

a) - Affectation par un pointeur sur le même type :

Soient P1 et P2 deux pointeurs sur le même type de données, alors l'instruction

P1 = P2;

fait pointer P1 sur le même objet que P2

b) - Addition et soustraction d'un nombre entier :

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

P+n pointe sur A[i+n]
P-n pointe sur A[i-n]

c) - 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]

d) Domaine des opérations :

L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies *à l'intérieur d'un tableau*. Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini.

Seule exception : Il est permis de 'pointer' sur le premier octet derrière un tableau (à condition que cet octet se trouve dans le même segment de mémoire que le tableau). Cette règle, introduite avec le standard ANSI-C, légalise la définition de boucles qui incrémentent le pointeur *avant* l'évaluation de la condition d'arrêt.

Exemples :

```
int A[10];
int *P;
P = A+9;    /* dernier élément -> légal */
P = A+10; /* dernier élément + 1 -> légal */
P = A+11; /* dernier élément + 2 -> illégal */
P = A-1;     /* premier élément - 1 -> illégal */
```

e) - 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 précède P1
- indéfini, si P1 et P2 ne pointent pas dans le même tableau

Plus généralement, la soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

f) - Comparaison de deux pointeurs :

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

La comparaison de deux pointeurs qui pointent *dans le même tableau* est équivalente à la comparaison des indices correspondants. (Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

3) Pointeurs et chaînes de caractères :

De la même façon qu'un pointeur sur **int** peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur **char** peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères. Un pointeur sur **char** peut en plus contenir *l'adresse d'une chaîne de caractères constante* et il peut même être *initialisé* avec une telle adresse.

A la fin de ce chapitre, nous allons anticiper avec un exemple et montrer que les pointeurs sont les éléments indispensables mais effectifs des fonctions en C.

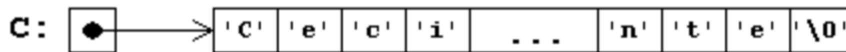
a) - Pointeurs sur char et chaînes de caractères constantes :

i)Affectation :

On peut attribuer l'adresse d'une chaîne de caractères constante à un pointeur sur **char** :

Exemple :

```
char *C;
C = "Ceci est une chaîne de caractères constante";
```



Nous pouvons lire cette chaîne constante (par exemple : pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible en ANSI-C.

ii)Initialisation :

Un pointeur sur **char** peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante :

```
char *B = "Bonjour !";
```

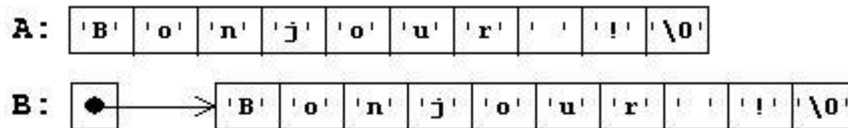
Attention !

Il existe une différence importante entre les deux déclarations :

```
char A[] = "Bonjour !";   /* un tableau   */
char *B  = "Bonjour !";   /* un pointeur */
```

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '\0'. Les caractères de la chaîne peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.



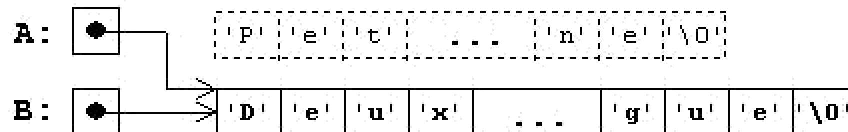
iii)Modification :

Si nous affectons une nouvelle valeur à un pointeur sur une chaîne de caractères constante, nous risquons de perdre la chaîne constante. D'autre part, un pointeur sur **char** a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur :

Exemple :

```
char *A = "Petite chaîne";
char *B = "Deuxième chaîne un peu plus longue";
A=B;
```

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue :

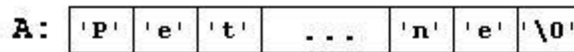


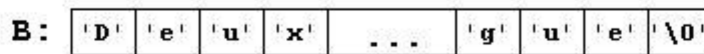
Attention !

Les affectations discutées ci-dessus ne peuvent pas être effectuées avec des tableaux de caractères :

Exemple :

```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus
longue"; char C[30];
A=B;                  /* IMPOSSIBLE -> ERREUR !!! */
C = "Bonjour !";      /* IMPOSSIBLE -> ERREUR !!! */
```





Dans cet exemple, nous essayons de copier l'adresse de B dans A, respectivement l'adresse de la chaîne constante dans C. Ces opérations sont impossibles et illégales parce que ***l'adresse représentée par le nom d'un tableau reste toujours constante***.

Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre (par exemple dans une boucle) ou déléguer cette charge à une fonction de <*stdio*> ou <*string*>.

<u>Conclusions :</u>

- Utilisons des *tableaux de caractères* pour déclarer les chaînes de caractères que nous voulons modifier.
- Utilisons des *pointeurs sur* ***char*** pour manipuler des chaînes de caractères constantes (dont le contenu ne change pas).
- Utilisons de préférence des *pointeurs* pour effectuer les manipulations à l'intérieur des tableaux de caractères.

4) <u>Pointeurs et tableaux à deux dimensions :</u>

L'arithmétique des pointeurs se laisse élargir avec *toutes* ses conséquences sur les tableaux à deux dimensions. Voyons cela sur un exemple :

<u>Exemple :</u>

Le tableau M à deux dimensions est défini comme suit :

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6,
                                    7, 8, 9},
        {10,11,12,13,14,15,16,17,18,19},

                  {20,21,22,23,24,25,26,27,28,29},
                  {30,31,32,33,34,35,36,37,38,39}};
```

Le nom du tableau M représente l'adresse du premier élément du tableau et pointe sur le ***tableau*** M[0] qui a la valeur :

```
{0,1,2,3,4,5,6,7,8,9}.
```

L'expression (M+1) est l'adresse du deuxième élément du tableau et pointe sur M[1] qui a la valeur :

```
{10,11,12,13,14,15,16,17,18,19}.
```

<u>Explication :</u>

Au sens strict du terme, un tableau à deux dimensions est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, le premier élément de la matrice M est le ***vecteur*** **{0,1,2,3,4,5,6,7,8,9}**, le deuxième élément est **{10,11,12,13,14,15,16,17,18,19}** et ainsi de suite.

L'arithmétique des pointeurs qui respecte automatiquement les dimensions des éléments conclut logiquement que :

M+I désigne l'adresse du tableau **M[I]**

Problème :

Comment pouvons-nous accéder à l'aide de pointeurs aux éléments de chaque composante du tableau, c.à-d. : aux éléments M[0][0], M[0][1], ... , M[3][9] ?

Discussion :

Une solution consiste à convertir la valeur de M (qui est un pointeur sur *un tableau du type* ***int****)*
en un pointeur sur ***int***. On pourrait se contenter de procéder ainsi :

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};

int *P;
P=M;
```

/* conversion automatique */Cette dernière affectation entraîne une conversion automatique de l'adresse &M[0] dans l'adresse &M[0][0]. (Remarquez bien que l'adresse transmise reste la même, seule la nature du pointeur a changé).

Cette solution n'est pas satisfaisante à cent pour-cent : Généralement, on gagne en lisibilité en explicitant la conversion mise en œuvre par l'opérateur de conversion forcée ("cast"), qui évite en plus des messages d'avertissement de la part du compilateur.

Solution :

Voici finalement la version que nous utiliserons :

```
int M[4][10] = {{0,1, 2, 3, 4, 5,6,7,8,9},
{10,11,12,13,14,15,16,17,18,19},
{20,21,22,23,24,25,26,27,28,29},
{30,31,32,33,34,35,36,37,38,39}};

int *P;
P = (int *)M;  /* conversion forcée */
```

Dû à la mémorisation ligne par ligne des tableaux à deux dimensions, il nous est maintenant possible traiter M à l'aide du pointeur P comme un tableau unidimensionnel de dimension 4*10.

Exemple :

Les instructions suivantes calculent la somme de tous les éléments du tableau M :

```
  int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6,
                                  7, 8, 9},
        {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},

                {30,31,32,33,34,35,36,37,38,39}};
int *P;
int I, SOM;
P = (int*)M;
SOM = 0;
for (I=0; I<40; I++)
     SOM += *(P+I);
```

Attention !

Lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel ***il faut calculer avec le nombre de colonnes indiqué dans la déclaration*** du tableau.

Exemple :

Pour la matrice A, nous réservons de la mémoire pour 3 lignes et 4 colonnes, mais nous utilisons seulement 2 lignes et 2 colonnes :

```
int A[3][4];
A[0][0]=1;
A[0][1]=2;
A[1][0]=10;
A[1][1]=20;
```

Dans la mémoire, ces composantes sont stockées comme suit :

L'adresse de l'élément A[I][J] se calcule alors par :

A+I***4**+J

Conclusion :

Pour pouvoir travailler à l'aide de pointeurs dans un tableau à deux dimensions, nous avons besoin de quatre données :

a) l'adresse du premier élément du tableau converti dans le type simple des éléments du tableau

b) la longueur d'une ligne réservée en mémoire (- voir déclaration - ici : 4 colonnes)

c) le nombre d'éléments effectivement utilisés dans une ligne (- p.ex : lu au clavier - ici : 2 colonnes)

d) le nombre de lignes effectivement utilisées (- p.ex : lu au clavier - ici : 2 lignes)