

# Les pointeurs

Modes d'adressage de variables. Définition d'un pointeur. Opérateurs de base. Opérations élémentaires. Pointeurs et tableaux. Pointeurs et chaînes de caractères. Pointeurs et enregistrements. Tableaux de pointeurs. Allocation dynamique de la mémoire. Libération de l'espace mémoire.

# 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 :

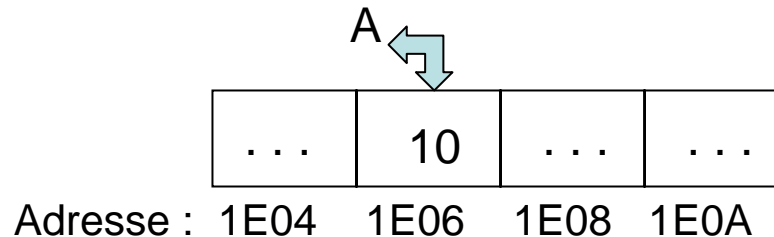
Les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions.
- Le traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs.
- Les pointeurs nous permettent de définir de nouveaux types de données : 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.

# 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.

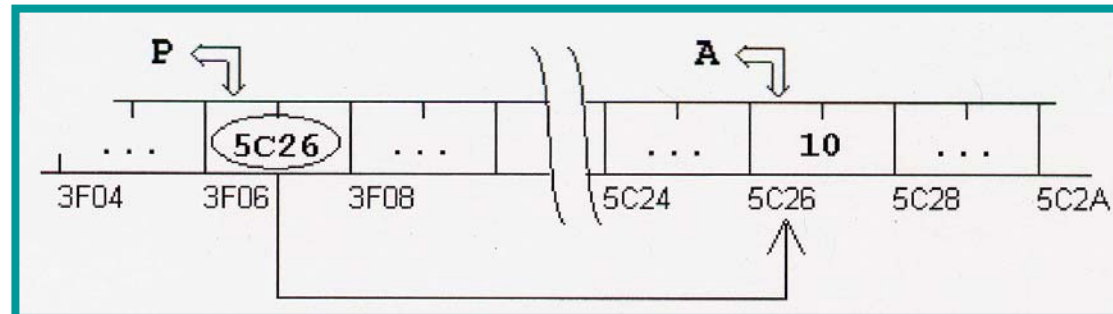
# Mode d'adressage indirect des variables

## 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 :



# 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`).

# 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.

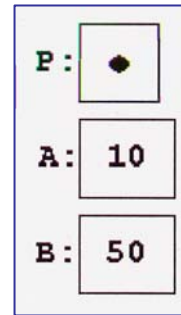
# Comment accéder au contenu d'une adresse ?

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

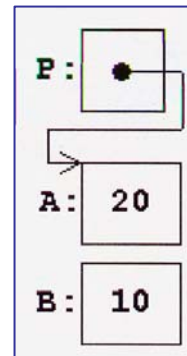
Syntaxe permettant d'avoir accès au contenu d'une adresse :

`* 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` (ex. : `cin >> *P;`).

# 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++
```

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



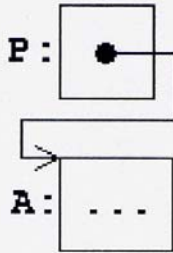
# Le pointeur NULL

- 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)

$A == *P \Leftrightarrow P == \&A$

$A == *\&A$  et  $P == \&*P$

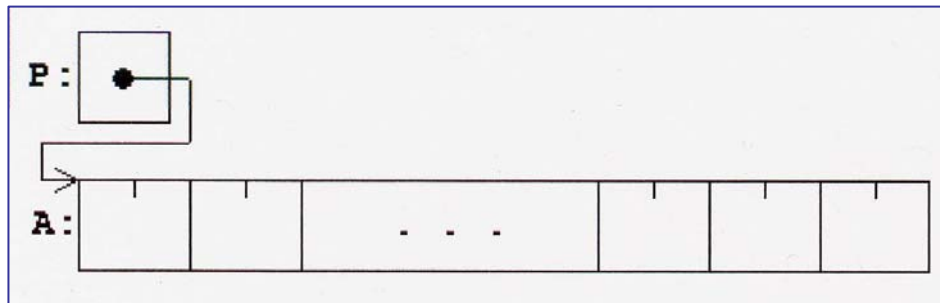
# Pointeurs et tableaux

- 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];
```



# 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<sup>ème</sup> composante à droite de \*P.

P - i pointe sur la i<sup>ème</sup> 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.

# Calcul d'adresse des composantes d'un tableau

**Note :** Il peut paraître surprenant que  $P + i$  n'adresse pas le  $i^{\text{ème}}$  octet après  $P$ , mais la  $i^{\text{ème}}$  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 # 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  $A$ , i.e. 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  $P$ .

# 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 précède 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.

## 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.



## 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  $A[i]$



# Copie des éléments positifs d'un tableau S dans un tableau T

```
#include <iostream.h>

void main()
{
    int S[10] = { -3, 4, 0, -7, 3 , 8, 0, -1, 4, -9};
    int T[10];
    int i, j;

    for (i = 0, j = 0; i < 10; i++)
        if (*(S + i) > 0)
        {
            *(T + j) = *(S + i);
            j++;
        }

    for (i = 0; i < j; i++) cout << *(T + i) << " ";
    cout << endl;
}
```

# Rangement des éléments d'un tableau dans l'ordre inverse

```
#include <iostream.h>
void main()
{
    int N;
    int tab[50];
    int somme = 0;
    cout << "Entrez la dimension N du tableau : ";
    cin >> N;
    for (int i = 0; i < N; i++)
    {
        cout << "Entrez la " << i << " ieme composante : ";
        cin >> *(tab+i);
        somme += *(tab+i);
    }
    for (int k = 0; k < N / 2; k++)
    {
        int echange = *(tab+k);
        *(tab+k) = *(tab + N - k - 1);
        *(tab + N - k - 1) = echange;
    }
}
```

## Rangement des éléments d'un tableau dans l'ordre inverse

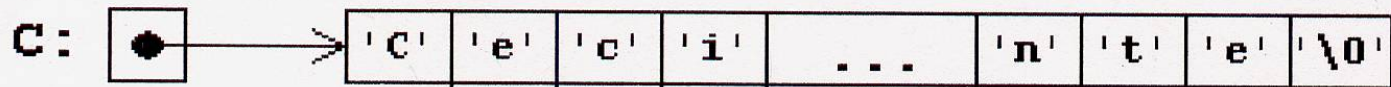
```
cout << endl << endl << "Affichage du tableau inverse." << endl;
for (int j = 0; j < N; j++)
{
    if((j % 3) == 0) cout << endl;
    cout << "tab[ " << j << " ] = " << *(tab+j) << "\t";
}
}
```

# Pointeurs et chaînes de caractères

- Tout ce qui a été mentionné concernant les pointeurs et les tableaux reste vrai pour les pointeurs et les chaînes de caractères.
- En plus, un pointeur vers une variable de type char peut aussi contenir l'adresse d'une chaîne de caractères constante et peut même être initialisé avec une telle adresse.

## *Exemple*

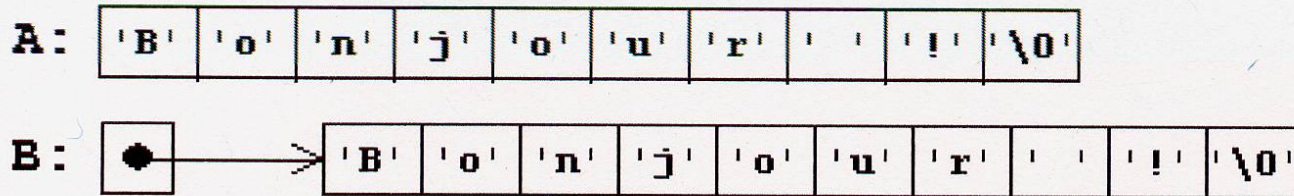
```
char *C;  
C = "Ceci est une chaîne de caractères constante";
```



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

# Distinction entre un tableau et un pointeur vers une chaîne constante

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



**A** a exactement la grandeur pour contenir la chaîne de caractères et \0.  
Les caractères peuvent être changés mais **A** va toujours pointer sur la même  
adresse en mémoire (pointeur constant).

## 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 !!! */
```

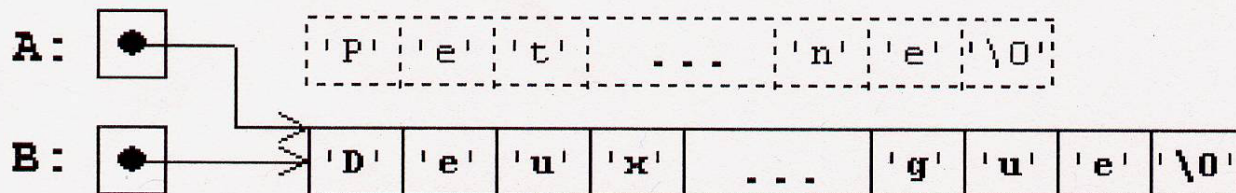
# Distinction entre un tableau et un pointeur vers une chaîne constante

**B** pointe sur une chaîne de caractères constante. Le pointeur peut être modifié et pointer sur autre chose (la chaîne de caractères constante originale sera perdue). La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée (`B[1] = 'o';` est illégal).

## 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:



Un pointeur sur `char` a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur.

# Avantage des pointeurs sur char

- Un pointeur vers char fait en sorte que nous n'avons pas besoin de connaître la longueur des chaînes de caractères grâce au symbole \0.
- Pour illustrer ceci, considérons une portion de code qui copie la chaîne CH2 vers CH1.

```
char * CH1;  
char * CH2;  
...
```

1<sup>ère</sup> version :

```
int I;  
I=0;  
while ((CH1[I]=CH2[I]) != '\0')  
    I++;
```

2<sup>ième</sup> version :

Un simple changement de notation nous donne ceci :

```
int I;  
I=0;  
while ((* (CH1+I)=* (CH2+I)) != '\0')  
    I++;
```

# Avantage des pointeurs sur char

Exploitions davantage le concept de pointeur.

```
while ((*CH1=*CH2) != '\0')
{
    CH1++;
    CH2++;
}
```

Un professionnel en C obtiendrait finalement :

```
while (*CH1++ = *CH2++)
    ;
```



# Pointeurs et tableaux à deux dimensions

```
Soit    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}};
```

M représente l'adresse du 1<sup>e</sup> élément du tableau et pointe vers le tableau M[0] dont la valeur est : { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. De même, M + i est l'adresse du i<sup>ème</sup> élément du tableau et pointe vers M[i] dont la valeur est la i<sup>ème</sup> ligne de la matrice.

```
cout << (*(M+2))[3];           // 23
```

## Explication :

Un tableau 2D est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, M + i désigne l'adresse du tableau M[i].

## Question :

Comment accéder à l'aide de pointeurs uniquement à une composante M[i][j] ?

Il s'agit de convertir la valeur de M qui est un pointeur sur un tableau de type `int` en un pointeur de type `int`.

# Pointeurs et tableaux à deux dimensions

Solution :

```
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 */
```

Puisque le tableau 2D est mémorisé ligne par ligne et que cette dernière affectation entraîne une conversion de l'adresse `&M[0]` à `&M[0][0]` \*, il nous est maintenant possible de traiter M à l'aide du pointeur P comme un tableau unidimensionnel de dimension 40.

- \* P et M renferme la même adresse mais elle est interprétée de deux façons différentes.

# Pointeurs et tableaux à deux dimensions

Exemple : Calcul de la somme de tous les éléments du tableau 2D 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);
```

**Note :** Dans cet exemple, toutes les lignes et toutes les colonnes du tableau sont utilisées. Autrement, on doit prendre en compte

- le nombre de colonnes réservé en mémoire,
- le nombre de colonnes effectivement utilisé dans une ligne,
- le nombre de lignes effectivement utilisé.

# 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.
```

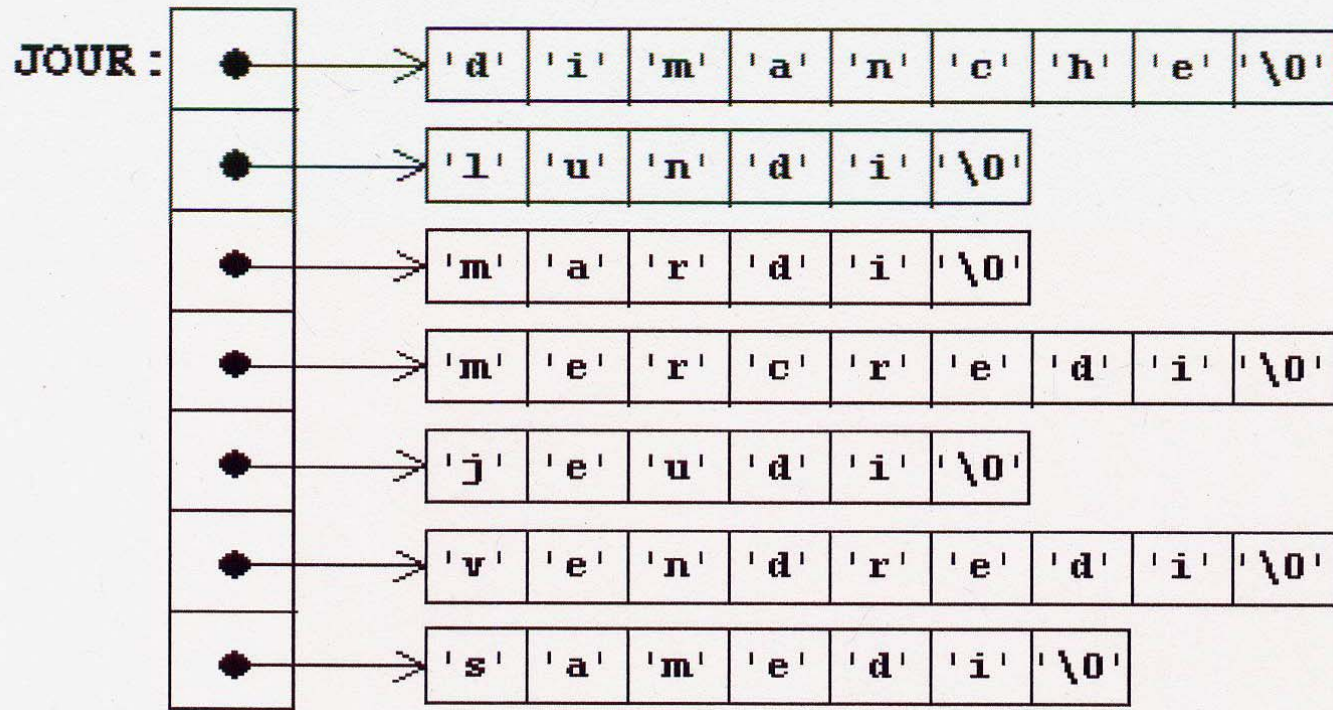
## Tableaux de pointeurs vers des chaînes de caractères de différentes longueurs

### 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.

# Tableaux de pointeurs



Affichage :

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

Pour afficher la 1<sup>e</sup> lettre de chaque jour de la semaine, on a :

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



# Tableaux de pointeurs

Si  $D[j]$  pointe dans un tableau,

$D[i]$	désigne l'adresse de la première composante
$D[i] + j$	désigne l'adresse de la j-ième composante
$*(D[i] + j)$	désigne le contenu de la j-ième composante

Les tableaux de pointeurs vers des chaînes de caractères de différentes longueurs sont d'un grand intérêt mais ce n'est pas le seul (à suivre).

# Allocation statique de la mémoire

- Jusqu'à maintenant, la déclaration d'une variable entraîne automatiquement la réservation de l'espace mémoire nécessaire.
- Le nombre d'octets nécessaires était connu au temps de compilation; le compilateur calcule cette valeur à partir du type de données de la variable.

## Exemples d'allocation statique de la mémoire

```
float A, B, C;           /* réservation de 12 octets */
short D[10][20];        /* réservation de 400 octets */
char E[] = {"Bonjour !"};
                        /* réservation de 10 octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                        /* réservation de 40 octets */
```

Il en est de même des pointeurs ( $p = 4$ ).

```
double *G;              /* réservation de p octets */
char *H;                /* réservation de p octets */
float *I[10];           /* réservation de 10*p octets */
```

# Allocation statique de la mémoire

Il en est de même des chaînes de caractères constantes ( $p = 4$ ).

## *Exemples*

```
char *J = "Bonjour !";  
        /* réservation de p+10 octets */  
float *K[] = {"un", "deux", "trois", "quatre"};  
        /* réservation de 4*p+3+5+6+7 octets */
```



# 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.
```

# 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>
```

# La fonction malloc et l'opérateur sizeof

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));
```

# La fonction malloc et l'opérateur sizeof

## Note :

S'il n'y a pas assez de mémoire pour satisfaire une requête, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande `exit` de `stdlib` et de renvoyer une valeur non nulle comme code d'erreur.

## Exemple :

Lire 10 phrases au clavier et ranger le texte.  
La longueur maximale d'une phrase est fixée à 500 caractères.

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

main()
{
    /* Déclarations */
    char INTRO[500];
    char *TEXTE[10];
    int I;
    /* Traitement */
    for (I=0; I<10; I++)
    {
```

# La fonction malloc et l'opérateur sizeof

```
gets(INTRO);
/* Réserve de la mémoire */
TEXTE[I] = malloc(strlen(INTRO)+1);
/* S'il y a assez de mémoire, ... */
if (TEXTE[I])
    /* copier la phrase à l'adresse */
    /* fournie par malloc, ... */
    strcpy(TEXTE[I], INTRO);
else
{
    /* sinon quitter le programme */
    /* après un message d'erreur. */
    printf("ERREUR: Pas assez de mémoire \n");
    exit(-1);
}
}
return 0;
}
```

# Ex. : Matrice triangulaire inférieure (partie I)

Exemple :

$$\begin{pmatrix} 12 & & & \\ -2 & 4 & & \\ 9 & -17 & 50 & \\ -98 & 19 & 25 & 75 \end{pmatrix}$$

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int * M[9];
    int i, j;

    // Allocation dynamique de la mémoire.

    for (i = 0; i < 9; i++) M[i] = malloc((i+1) * sizeof(int));
```

## Ex. : Matrice triangulaire inférieure (partie I)

```
//      Initialisation de la matrice.

for (i = 0; i < 9; i++)
    for (j = 0; j <= i; j++)
        *(M[i] + j) = (i + 1) * 10 + j + 1;

//      Affichage des éléments de la matrice.

for (i = 0; i < 9; i++)
{
    for (j = 0; j <= i; j++)
        printf("\t%d", *(M[i] + j));
    for (j = i+1; j < 9; j++)
        printf("\t0");
    printf("\n");
}
}
```

Reprenons le même exemple où, cette fois, le # de lignes de la matrice est lu.

## Ex. : Matrice triangulaire inférieure (partie II)

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

```
void main()
{
```

```
    int ** M; 
    int N;
    int i, j;
```

```
    printf("Entrez le nombre de lignes de la matrice : ");
    scanf("%d", &N);
```

```
    //      Allocation dynamique de la mémoire.
```

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

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





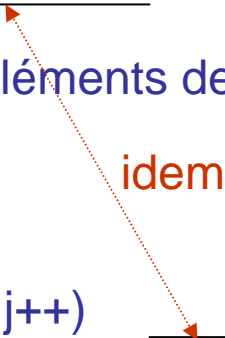
## Ex. : Matrice triangulaire inférieure (partie II)

```
//      Initialisation de la matrice.

for (i = 0; i < N; i++)
    for (j = 0; j <= i; j++)
        *((*(M+i)) + j) = (i + 1) * 10 + j + 1;

//      Affichage des éléments de la matrice.

for (i = 0; i < N; i++)
{
    for (j = 0; j <= i; j++)
        printf("\t%d", M[i][j]);
    for (j = i+1; j < N; j++)
        printf("\t0");
    printf("\n");
}
}
```



# Libération de l'espace mémoire

- 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.

# Allocation dynamique, libération de l'espace mémoire en C++

- Le mot clé `new` permet de réserver de l'espace selon le type de donnée fourni.

**Syntaxe :** `new type de donnée`

L'opérateur renvoie l'adresse du bloc de mémoire allouée.  
Si l'espace n'est pas disponible, l'opérateur renvoie 0.

**Exemple :** `unsigned short int * pPointeur;`  
`pPointeur = new unsigned short int;`

ou encore,

`unsigned short int * pPointeur = new unsigned short int;`

- On peut également affecter une valeur à cette zone. Ex. : `*pPointeur = 25;`
- Pour libérer l'espace alloué avec `new`, on utilise l'opérateur `delete` une seule fois.  
Ex.: `delete pPointeur;`

Autrement, il se produira une erreur à l'exécution. Pour éviter ceci, mettez le pointeur à 0 après avoir utilisé l'opérateur `delete`.

# Allocation dynamique, libération de l'espace mémoire en C++

- Libérer le contenu d'un pointeur nul est sans incidence.

```
int * p = new int;
delete p;           // libérer la mémoire.
p = 0;
...
delete p;           // sans incidence sur le programme.
```

- Réaffecter une valeur à un pointeur alors que celui-ci n'est pas nul génère une perte de mémoire.

```
unsigned short int * pPointeur = new unsigned short int;
*pPointeur = 72;
pPointeur = new unsigned short int;
*pPointeur = 36;
```

À chaque instruction `new` devrait correspondre une instruction `delete`.

- Tenter de libérer le contenu d'un pointeur vers une constante ou une variable allouée de manière statique est une erreur. Ex. :

```
const int N = 5;
int * p = &N;
delete p;
```

# Allocation dynamique, libération de l'espace mémoire en C++

- Comment libérer l'espace mémoire d'un tableau ?

```
float * p = new float[10];
```

```
...
```

```
delete [] p;
```

```
// libérer la mémoire.
```

```
p = 0;
```



car, nous sommes en présence d'un tableau.

# Matrice de réels - exemple

```
#include <iostream.h>

void main()
{
    // Saisie de la dimension de la matrice.

    int M, N;
    cout << "Nombre de lignes : ";
    cin >> M;
    cout << "Nombre de colonnes : ";
    cin >> N;

    // Construction et initialisation d'une matrice réelle M x N.

    typedef float * pReel;
    pReel * P;
    P = new pReel[M];
```

# Matrice de réels - exemple

```
for (int i = 0; i < M; i++)
{
    P[i] = new float[N];
    for (int j = 0; j < N; j++)    P[i][j] = (float) 10*i + j;
}

// Affichage d'une matrice M x N.

for (i = 0; i < M; i++)
{
    cout << endl;
    for (int j = 0; j < N; j++)    cout << P[i][j] << " ";
}
cout << endl;

// Libération de l'espace.

for (i = 0; i < M; i++)    delete [] P[i];
delete [] P;
}
```



# Pointeur générique

Une variable de type `void *` est un pointeur générique capable de représenter n'importe quel type de pointeur.

```
#include <stdio.h>
void main()
{
    int A = 5;          void * P = &A;
    if ((* (int *) P) == 5) // *P est invalide car on ne connaît
        printf("%d", (*(int *) P)); // pas le type pointé par P.
}
```

On peut affecter un pointeur à un autre si les 2 sont de même type. S'ils ne le sont pas, il faut effectuer une conversion explicite.

La seule exception est le type `void *`. On ne peut toutefois pas affecter un pointeur `void *` directement à un pointeur d'un autre type.

```
#include <iostream.h>
void main()
{
    void * P;          float * R;          int Q = 5;
    P = &Q;           R = (float *)P;
    cout << *R;      // Donne des résultats erronés. }
}
```

# Usage de const avec les pointeurs

- Un pointeur constant est différent d'un pointeur à une constante.

```
int n = 44;  
int* p = &n;  
++(*p);
```

```
++p;
```

```
int* const cp = &n;
```

un pointeur constant

```
++(*cp);
```

```
++cp;
```

illégal

```
const int k = 88;
```

```
const int * pc = &k;
```

un pointeur à une constante

```
++(*pc);
```

illégal

```
++pc;
```

```
const int* const cpc = &k;
```

un pointeur constant  
à une constante

```
++(*cpc);
```

illégal

```
++cpc;
```

illégal

# Avant-goût des structures de données

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    // Déclaration des types de données.
    struct Fiche_etudiant
    {
        char nom[25];
        char prenom[25];
        int age;
        bool sexe;
        int matricule;
        struct Fiche_etudiant * suivant;
    };

    struct Fiche_etudiant * pEnsemble_des_fiches = NULL;
    struct Fiche_etudiant * pointeur = NULL;

    int Nombre_de_fiches = 0;
    char test;
```

# Avant-goût des structures de données

```
// Saisie des fiches des étudiants.
```

```
for (Nombre_de_fiches = 0; Nombre_de_fiches < 50; Nombre_de_fiches++)  
{
```

```
    printf("\nVoulez-vous entrer les donnees d'une %s fiche (O ou N) ? ",  
          Nombre_de_fiches ? "autre " : "");
```

```
    scanf(" %c", &test);
```

```
    if(test == 'N' || test == 'n') break;
```

```
    pointeur=(struct Fiche_etudiant *) malloc(sizeof(struct Fiche_etudiant));
```

```
    (*pointeur).suivant = pEnsemble_des_fiches;
```

```
    pEnsemble_des_fiches = pointeur;
```

```
    scanf("%s%s%i%i%i",
```

```
          (*pEnsemble_des_fiches).nom,  
          (*pEnsemble_des_fiches).prenom,  
          &(*pEnsemble_des_fiches).age,  
          &(*pEnsemble_des_fiches).sexe,  
          &(*pEnsemble_des_fiches).matricule);
```

```
}
```

# Avant-goût des structures de données

// Affichage de l'ensemble des fiches d'étudiants.

```
pointeur = pEnsemble_des_fiches;
while (pointeur != NULL)
{
    printf("\n%s %s %i %i %i ", (*pointeur).nom,
        (*pointeur).prenom,
        (*pointeur).age,
        (*pointeur).sexe,
        (*pointeur).matricule);
    pointeur = (*pointeur).suivant;
}
}
```

La structure de données utilisée est une pile.