# Notions de Base

## Allocation mémoire & Langage C

Hidouci W.K. / ESI / ALSDD / 2017/2018

Une **variable** déclarée possède :

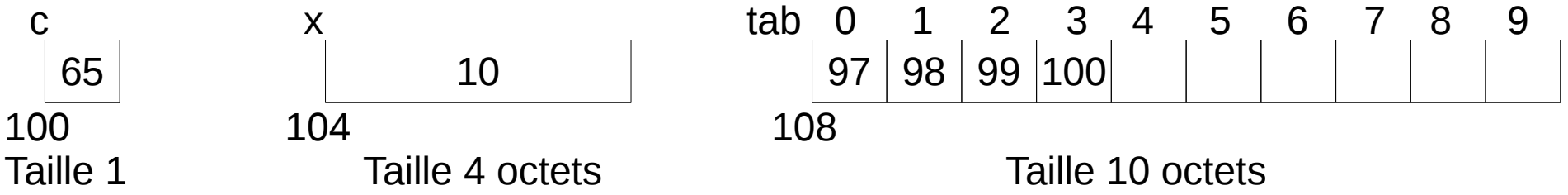
Un nom (identificateur)

Une zone qui lui est réservée en mémoire

→ caractérisée par l'adresse de début et la taille

Exemples de déclarations avec initialisations :

```
char c = 'A' ;   /* le code ascii de 'A' est l'octet 65 */
int x = 10 ;
char tab[10] = {'a', 'b', 'c' , 'd' } ;
      /* codés :  97, 98, 99, 100 */
```



Un **pointeur** est variable pouvant contenir des adresses

Exemple:

```
int x ;      /* x une variable de type entier */
int *p ;     /* p une variable de type pointeur vers entier */
```


```
p = &x;      /* affecte à p l'adresse de x, on dit alors : p pointe x */
```


```
*p = 10 ;    /* affecte au contenu de p (donc à x) la valeur 10 */
```

Relation entre **pointeurs** et **tableaux** en langage C

La déclaration d'un tableau :
*type_element tab[ MAX ] ;*
crée une variable de taille *MAX * sizeof(type_element)* octets.
L'identificateur *tab* représente alors une expression constante de type pointeur vers *type_element*.
C'est l'adresse du 1er élément du tableau

Exemples:

```
int tab[10] ;        /* x est un tableau de 10 entiers */
int *p ;             /* p est un pointeur vers entier */

p = tab ;            /* p contient l'adresse du 1er élt de tab : &tab[0] */

*p = 10 ;            /* ou: *tab = 10 ou tab[0] = 10  ou p[0] = 10 */
*(p+1) = 20 ;        /* ou: *(tab+1) ou tab[1] ou p[1] = 20 */

p = tab + 3 ;        /* p pointe maintenant le 4e élt de tab : &tab[3] */
*p = 30 ;            /* équivalent à : tab[3] = 30 ou p[0] = 30 … */

p[2] = 40 ;          /* équivalent à : tab[5] = 40 … */
```

```c
#include <stdio.h>

int main( void )
{
    int tab[10];
    int *p, i;

    p = tab;          // équivalent à p = &tab[0]

    *p  = 10;         // équivalent à tab[0] = 10
    *(p+1) = 20;      // équivalent à tab[1] = 20
    p[2] = 30;        // équivalent à tab[2] = 30
    tab[3] = 40;
    *(tab+4) = 50;    // équivalent à tab[4] = 50

    printf("Les adresses sont affichés en hexadécimal\n\n");
    printf("Parcours du tableau avec (@base+deplcaement):\n");
    printf("Adr_de_base \t\t Deplacement \t Contenu\n");
    for (i=0; i<5; i++)
        printf("tab:%p  \t i:%d  \t\t *(tab+%d):%d\n", tab, i, i, *(tab+i) );

    printf("\nParcours du tableau avec incrémentation du pointeur p:\n");
    printf("Adresse \t\t Contenu\n");
    for (i=0; i<5; i++) {
        printf("p:%p \t *p:%d\n", p, *p);
        p++;      // passer au prochain élément (par pas de sizeof(int) octets )
    }

    return 0;
}
```

## Exemple de résultats affichés par le programme précédent

Les adresses sont affichés en hexadécimal

Parcours du tableau avec (@base+deplcaement):

| Adr_de_base | Deplacement | Contenu |
|---|---|---|
| tab:0x7ffe44afe7a0 | i:0 | *(tab+0):10 |
| tab:0x7ffe44afe7a0 | i:1 | *(tab+1):20 |
| tab:0x7ffe44afe7a0 | i:2 | *(tab+2):30 |
| tab:0x7ffe44afe7a0 | i:3 | *(tab+3):40 |
| tab:0x7ffe44afe7a0 | i:4 | *(tab+4):50 |

Parcours du tableau avec incrémentation du pointeur p:

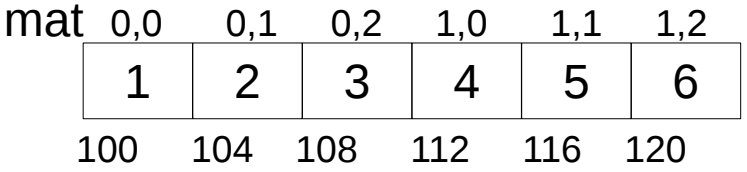
| Adresse | Contenu |
|---|---|
| p:0x7ffe44afe7a0 | *p:10 |
| p:0x7ffe44afe7a4 | *p:20 |
| p:0x7ffe44afe7a8 | *p:30 |
| p:0x7ffe44afe7ac | *p:40 |
| p:0x7ffe44afe7b0 | *p:50 |

# Tableaux à plusieurs dimensions

Un tableau à n dimensions (n>1) est un tableau à une dimension où chaque élément est un tableau à n-1 dimensions

Exemple d'un tableau à 2 dimensions (2 lignes et 3 colonnes) :
int mat[2][3] = { {1,2,3} , {4,5,6} } ;  // ou alors { 1,2,3,4,5,6 }

*mat* est l'adresse d'une zone mémoire contenant les éléments rangés ligne par ligne




Taille = 6 entiers = 6*4 = 24 octets
(de l'octet 100 à l'octet 123)
En supposant qu'un entier occupe 4 octets

mat[i][j] représente l'élément se trouvant à la ligne i et la colonne j

Pour accéder à l'élément d'indice [ i ] [ j ] dans un tableau de NL lignes et NC colonnes,
le compilateur effectue le calcul suivant :

**@[ i ][ j ] = @base + sizeof(typeElmt) * (i * (NC) + j )**

Dans l'exemple ci-dessus, l'adresse de mat[1][1] = 100 + 4*(1*3 + 1) = 116

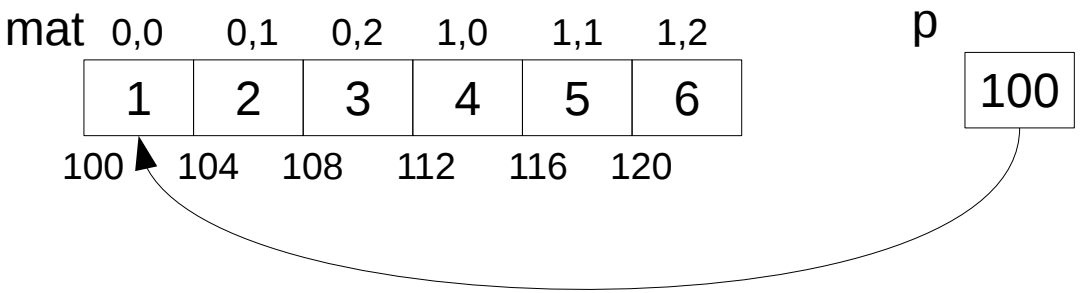
Dans le cas général d'un tableau à k dimensions : $\mathbf{N_1}$, $\mathbf{N_2}$, … $\mathbf{N_k}$

$$@[i_1][i_2]...[i_k] = @base + sizeof(typeElmt) * (i_1 * N_2 * N_3 ... * N_k + i_2 * N_3 * N_4 ... * N_k + \ldots + i_{k-1} * N_k + i_k )$$

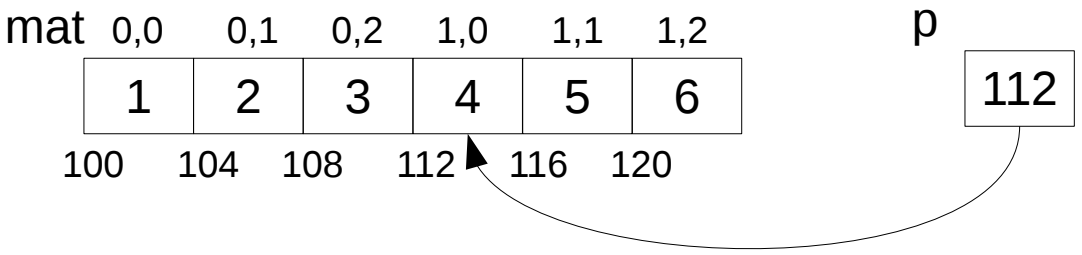
# Tableaux à plusieurs dimensions

On peut aussi utiliser les pointeurs pour manipuler les tableaux à plusieurs dimensions

Exemple d'un tableau à 2 dimensions (2 lignes et 3 colonnes) :
int mat[2][3] = { 1,2,3,4,5,6 } ;
int (*p)[3] ; // p est un pointeur vers un tableau de 3 entiers
// donc p peut être vu comme un tableau de tableau de 3 entiers
p = mat ;
// alors *(p+1) ou p[1] ou mat[1] représentent le tableau de la 2e ligne de mat : {4,5,6}
// et (*(p+1))[0] ou p[1][0] ou mat[1][0] représentent le 1er élément de la 2e ligne (mat[1][0]) : 4




p = mat+1 ;
// alors *p ou p[0] ou mat[1] représentent aussi le tableau de la 2e ligne de mat : {4,5,6}
// et (*p)[0] ou p[0][0] ou mat[1][0] représentent aussi le 1er élément de la 2e ligne (mat[1][0]) : 4



# Allocation dynamique

Il est possible de créer et détruire des variables durant l'exécution de programmes à l'aide de fonctions prédéfinies

En langage C, la bibliothèque standard (libc) contient quelques fonctions pour l'allocation mémoire

*#include <stdlib.h>*

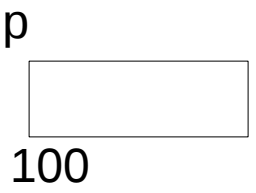
***malloc ( n )*** *: Alloue une zone mémoire de taille n octets et retourne son adresse*

***free( p )*** *: Libère (détruit) la zone mémoire pointée par p*

Il est donc possible de rajouter un nombre quelconque de variables dynamiquement durant l'exécution d'un programme

Ces nouvelles variables n'ont pas de nom, on ne peut les manipuler qu'à travers leurs adresses

Exemple :
int *p ;

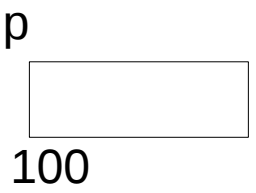



p = malloc( sizeof(int) ) ;  /* allocation d'une variable de type entier */


/* p contient maintenant l'adresse d'une nouvelle variable dynamique de type entier (cette nouvelle variable se trouve à l'adresse 200) */

*p = 10 ; /* affecte à cette variable dynamique la valeur 10 */


/* Détruit la variable dynamique */
free(p) ;



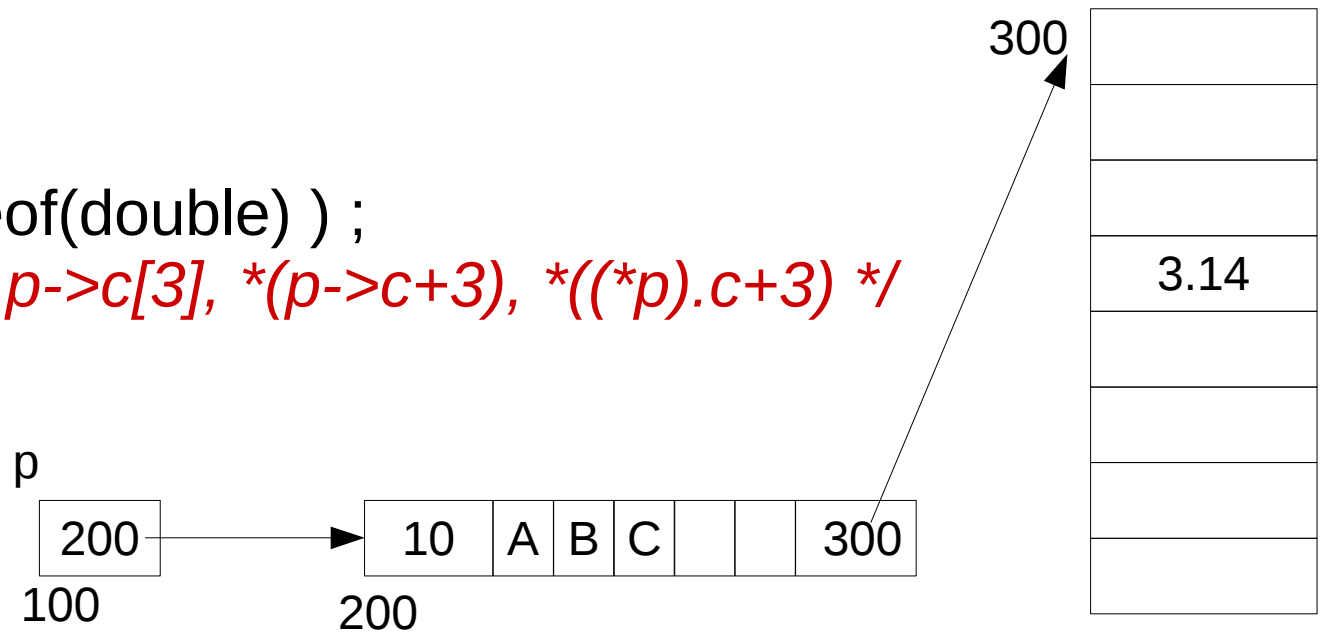
# Les variables dynamiques peuvent être de n'importe quel type

Exemple : (voir aussi fichier source '**struct.c**')

```
struct enreg {
  int a ;
  char b[5] ;
  double *c ;
} *p ;

p = malloc( sizeof( struct enreg ) ) ;

(*p).a = 10 ;     /* on peut aussi écrire p->a = 10 */
(*p).b[0] = 'A' ;
(*p).b[1] = 'B' ;
(*p).b[2] = 'C' ;
(*p).c = malloc( 8 * sizeof(double) ) ;
(*p).c[3] = 3.14 ; /* ou : p->c[3], *(p->c+3), *((*p).c+3) */
...
```



## Les chaînes de caractères

C'est des tableaux de caractères où la fin de chaîne est le caractère '\0'

Dans ce cas on pourra utiliser les fonctions prédéfinies sur les chaînes de caractères dans la libc (#include <string.h>)
→ strlen(ch) : longueur, strcpy(d,s) : affectation de chaînes, ...

Exemples :

```
char t[ ] = "abc" ; // ou alors : char t[ ] = {'a' , 'b', 'c', '\0' } ;
// t est un tableau de 4 caractères initialisé par la chaîne "abc"

char *p = "abc";
// même chose, sauf que la chaîne n'est pas
// modifiable car c'est une constante
```

(voir aussi fichier source '**chaine.c**')