الكلية متعددة التخصصات - ورزازات

ⵜⴰⵖⵡⴰⵙⵜ ⵜⴰⵣⵜⵣⴰⵡⵉⵜ – ⵡⵓⵔⵣⵣⵓⵜ

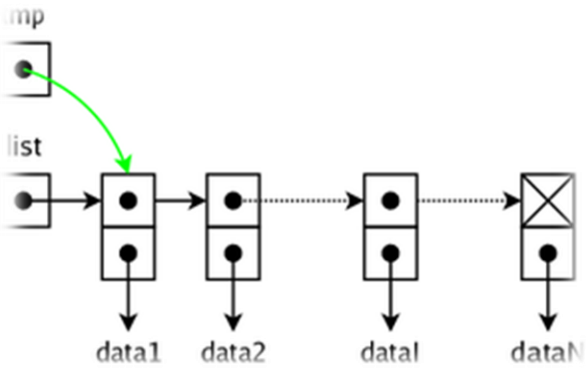
FACULTÉ POLYDISCIPLINAIRE DE OUARZAZATE

Sciences Mathématiques et informatique

Semestre 4

Support de cours

M22 : Structures de données en langage C



Professeur : Abdelhadi Bouain

Table de matière

Rappel

Un exemple de programme en langage C

Ce programme permet de lire deux nombres réels et de réaliser leur somme, soustraction ou multiplication selon le choix de l'utilisateur.

```c
#include <stdio.h>
main()
{
    /* déclaration des variables */
    float  x, y, resultat;
    char op;

    printf("Entrez le premier nombre : \n");
    scanf("%f", &x);            /* Entrer le premier nombre. */
    printf("Entrez le deuxième nombre : \n");
    scanf("%f", &y);            /* Entrer le deuxième nombre. */
    printf("Entrez l'opération + , - ou  *  : \n");
    op = getchar () ;           /* Entrer l'opération */

    if (op == '+')   /* vérifier si l'opération est une addition*/
                {
                            resultat = x+y;
                            printf ("la somme des deux nombres: %f ", resultat) ;
                }
         else if (op == '-') /* vérifier si l'opération est une soustraction*/
                {
                            resultat = x-y;
                            printf ("la soustraction des deux nombres: %f ", resultat) ;
                }
              else  /* Si c'est pas une addition ou soustraction alors réaliser la multiplication*/
                  {
                           resultat = x*y;
                            printf ("la multiplication des deux nombres: %f ", resultat) ;
                  }
}
```

Un deuxième exemple de programme en langage C

Ce programme permet de chercher la plus grande valeur (max) dans un tableau.

```c
#include <stdio.h>
main()
    {
      // Déclaration des variables
        int i, nbr, tab[100], max;

      /* Définition du nombre d'éléments à insérer dans le tableau*/
        printf("Entrez le nombre total d'éléments: ");
        scanf("%d", &nbr);

        // Stocker les nombres saisis par l'utilisateur
        for(i = 0; i < nbr; ++i)
        {
           printf("Entrer le nombre %d: ", i+1);
           scanf("%d", &tab[i]);
        }

        //considérer le premier élément comme max
        max = tab[0];

        // Boucle pour chercher le max dans le reste du tableau
        for(i = 1; i < nbr; i++)
        {
           if(max < tab[i])
                      {
                           max = tab[i];
                      }
        }

        printf("Le plus grand élément est %d", max);
     }
```

1) Chapitre I : Pointeurs et allocation dynamique de la mémoire.

1.1 - Introduction.

Toute variable dans un programme occupe un espace dans la mémoire centrale (La RAM). La taille de l'espace mémoire qu'occupe une variable dépend de son type.

Le tableau suivant résume la taille de chaque type de variable en C :

Type de donnée	Signification	Taille (en octets)
Char	***Caractère***	***1***
unsigned char	Caractère non signé	1
short int	Entier court	2
unsigned short int	Entier court non signé	2
Int	***Entier***	*2 ou **4***
unsigned int	Entier non signé	2 ou 4
long int	*Entier long*	*4*
unsigned long int	Entier long non signé	4
Float	***Flottant (réel)***	***4***
Double	***Flottant double***	***8***
long double	Flottant double long	10

Remarque :

> Pour identifier la taille d'une variable, vous pouvez utiliser l'opérateur **sizeof()** qui fournit la taille d'une variable en octets.

● **Exemple**:

```
main()
{
        int a;
        long int b;
        float c;
        long double d;

        printf("la taille d'un entier : %d  \n", sizeof(a));
        printf("la taille d'un long entier : %d  \n", sizeof(b));
        printf("la taille d'un réel : %d  \n", sizeof(c));
        printf("la taille d'un long réel : %d  \n", sizeof(d));

}
```

● **Résultat**

```
la taille d'un entier : 4
la taille d'un long entier : 4
la taille d'un reel : 4
la taille d'un long reel : 12
```

Vous pouvez aussi utiliser directement **sizeof** avec un type, par exemple :

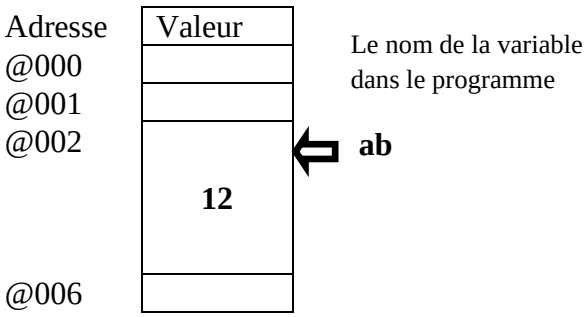
- **sizeof(int)** vaudra 4,
- **sizeof(double)** vaudra 8.

1.2 Adressage direct

Chaque variable a un ***type***, un ***identificateur (nom), une adresse*** et ***une valeur***.

● Exemple:

```
int ab ;
ab= 12;
```



La mémoire centrale

Ce programme permet de créer une variable **de type int** (lui réserver 4 octets dans la mémoire), cette variable est **nommée ab**. **La valeur** de ab est 12.

```
printf("% d", ab) ;
```

Permet d'afficher la valeur de la variable **ab**.

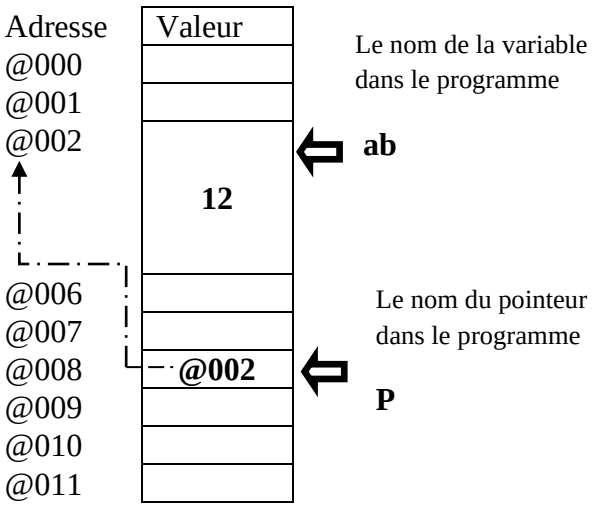
- Le principe de l'adressage direct est de manipuler une variable par son nom.

L'adressage direct *: L'*accès au contenu d'une variable par **le nom de la variable**.

1.3 Adressage indirect

Le principe de ***l'adressage indirect*** est de manipuler une variable **par un pointeur qui** contient l'adresse de cette variable.

●Exemple:



La mémoire centrale

Au lieu d'utiliser le nom de la variable ab en utilisera le pointeur P pour accéder à son contenu. La déclaration et l'utilisation des pointeurs sont traitées dans la section suivante.

1.4 Notion de pointeur.

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

1.5 Déclaration d'un pointeur.

```
int * P ;
```

Déclarer une variable nommée P comme étant un pointeur sur des entiers.
Le pointeur P contiendra uniquement l'adresse d'un entier.

1.6 Affecter une adresse a un pointeur.

```
P = &ab ;
```

L'opérateur **&**ab représente l'adresse de ab. Après cette instruction P contiendra l'adresse de la variable ab, on dit que **P pointe sur ab.**

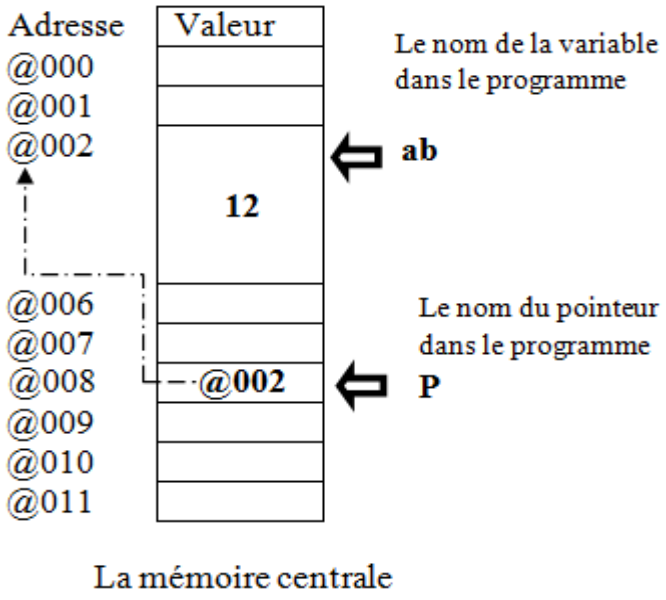
Un pointeur **doit toujours être initialisé** par une adresse ou bien une valeur NULL.

```
P = NULL ;
```

Par convention, un pointeur de valeur NULL ne pointe sur rien.

● **Exercice:** Créer un pointeur P sur une variable de type entier.

```
int ab ;
ab= 12 ;
int *P ;
P=&ab ;
```



1.7 Modifier la valeur d'une variable en utilisant un pointeur.

```
int ab ;
ab= 12 ;
int *P ;
P=&ab ;
*P = 86 ;
```

L'opérateur ***** permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si P est un pointeur vers un entier ab, *P désigne la valeur de ab.

- **P** est équivalent à **&ab.**
- ***P** est équivalent à **ab.**

1.8 Quelques exemples.

❖ **Exemple 1** : Afficher l'adresse d'une variable.

```
int  a;
float b ;
printf("L'adresse de  a:   %x   \n",   &a  );
 printf("L'adresse de b:   %x  \n",    &b  );
```

❖ **Résultat**

```
L'adresse de  a:   28ff44
L'adresse de  b:   28ff40
```

❖ **Exemple 2** : Déclarer des pointeurs sur différents types.

```
int    *ip;            / * Pointeur sur un entier * /
double *dp;                  /* Pointeur sur un double */
float  *fp;                  /* Pointeur sur un réel*/
char   *ch                   /* Pointeur sur un caractère */
```

❖ **Exemple 3** : Déclarer un pointeur et afficher la valeur de la variable sur laquelle il pointe.

```
int a = 3;
int *p;
p = &a;
printf("Le contenu pointé par P est  = %d \n", *p);
```

Résultat :

```
Le contenu pointé par P est : 3
```

❖ **Exemple 4 :**

```
int *P, X ;
P = &X;
```

Les expressions suivantes sont équivalentes:

P	⇔	**&X**
***P**	⇔	**X**
Y = *P+1	⇔	**Y = X+1**
***P = *P+10**	⇔	**X = X+10**
***P += 2**	⇔	**X += 2**
++*P	⇔	**++X**
(*P)++	⇔	**X++**

Remarque :

- L'opérateur d'indirection * et d'adresse & ont une priorité **plus élevée** par rapport à l'addition, soustraction, multiplication et division.
- L'opérateur d'indirection * et d'adresse & ont une priorité **moins élevée** par rapport à la post-incrémentation ++ et la post-décrémentation –

❖ **Exemple 5 :** Soit les deux programmes suivants

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

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

Supposons qu'avant la dernière affectation de chacun de ces programmes, on est dans cette configuration :

objet	Adresse	Valeur
I	@1000	3
J	@1004	6
p1	@8000	@1000
p2	@9000	@1004

➢ Après l'exécution de la dernière instruction de chacun des deux programmes.

Après programme 1

objet	adresse	valeur
I	@1000	6
J	@1004	6
p1	@8000	@1000
p2	@9000	@1004

Après programme 2

objet	adresse	valeur
I	@1000	3
J	@1004	6
p1	@8000	@1004
p2	@9000	@1004

❖ **Exemple 6 :** Quelle est la valeur de a, b et c après exécution du programme suivant ?

```
main(){
    int a=1,b=2,c=3;
    int *P1,*P2;
    P1=&a;
    P2=&c;
    *P1=(*P2)++;
    P1=P2;
    P2=&b;
    *P1 -= *P2;
    ++*P2;
    P1 = &b;
    *P1 *= *P2;
    printf("A : %d, B: %d, C: %d", a, b, c);
}
```

Exemple d'exécution :

	A	B	C	P1	P2
int a=1, b=2, c=3;	1	2	3		
P1=&a				@A	
P2=&c					@C
*P1=(*P2)++	3		4		
P1=P2				@C	
P2=&b					@B
*P1 -= *P2			2		
++*P2		3			
P1 = &b				@B	
*P1 *= *P2		9			
	3	9	2	@B	@B

1.9 Arithmétique des pointeurs

Il est possible d'effectuer des opérations arithmétiques sur les pointeurs.

Les seules opérations valides sont les opérations d'addition et soustraction des entiers et la soustraction de pointeurs.

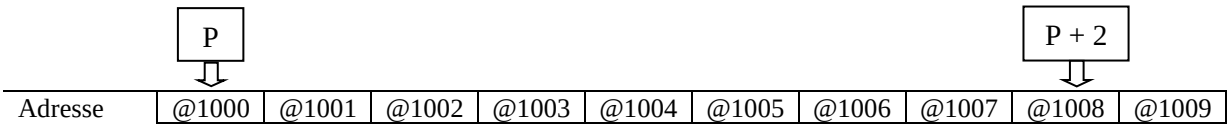
Elles sont définies comme suit :

> P + i = **adresse contenue dans** P + i * taille (élément pointé par p)

> P2 - P1 = (adresse contenue dans P2 - adresse contenue dans P1) / taille (éléments pointés par P1 et P2)

Exemple :

Si P est un pointeur sur les entiers et l'adresse contenue dans ce pointeur est 1000. Alors, P+2 est égal à l'adresse de P + 2 * taille d'un entier = 1000 + 2*4 = 10008.



1.10 Tableau simple et pointeurs.

En langage C, **l'identificateur (le nom) d'un tableau,** lorsqu'il est employé seul (sans indices à sa suite), est considéré comme **un pointeur (constant) sur le début du tableau.**

Dans la déclaration suivante :

```
int  t [10] ;
```

La notation **t** est alors totalement équivalente à **&t[0].**

Voici quelques exemples de notations équivalentes :

```
t+1                    &t [1]
t+i                    &t[i]
t[i]                   * (t+i)
```

- **Exemple 1 :** Pour remplir un tableau de 10 éléments par 1.

Méthode 1 : (sans utilisation de pointeur)

```
int i ;
for (i=0 ; i<10 ; i++)
t[i]  = 1 ;
```

Méthode 2 : (utilisation du nom du tableau comme pointeur)

```
int i ;
for (i=0 ; i<10 ; i++)
* (t+i) = 1 ;
```

Attention !!

Un nom de tableau est un pointeur constant. Autrement dit, dit la valeur de **t** ne doit pas changer. Une expression telle que **t= t+1** n'est pas valide.

- **Exemple 2 :**

```
int A[10];
int *P;
P=A ;
```

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+1) désigne le contenu de A [1]

*(P+2) désigne le contenu de A [2]

❖ Le programme suivant permet de remplir un tableau de 10 éléments par 1, en utilisant un pointeur.

```
int i ;
int A[10];
int *P;
P=A ;
for ( i=0 ; i<10 ; i++)
* (P+i) = 1 ;
```

Ou bien

```
int i ;
int A[10];
int *P;
P=A ;
for ( i=0 ; i<10 ; i++ , P++)
* P = 1 ;
```

❖ **Exercice**

Soit P un pointeur qui 'pointe' sur un tableau A:

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions:

***P+2**	14
***(P+2)**	34
&P+1	&P l'adresse du pointeur P dans la mémoire. &P + 1 l'adresse du pointeur suivant « à éviter - rarement utilisée »
&A[4]-3	&A[1]
A+3	&A[3]
&A[7]-P	7
P+(*P-10)	&A[2]
***(P+*(P+8)-A[7])**	23

1.11 Tableaux à deux dimensions et pointeurs.

- Déclaration d'un tableau à deux dimensions : (tableau de 3 lignes, 4 colonnes)

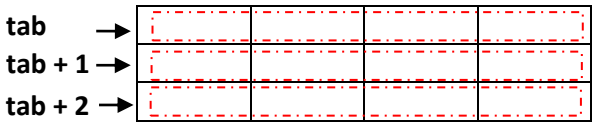
```
int tab[3] [4] ;
```

L'identificateur (le nom) d'un tableau, employé seul, représente l'adresse du début du tableau. Cependant, **il n'est plus du type int* (pointeur sur un entier) mais un pointeur sur un bloc de 4 entiers. Il s'agit donc en fait d'un pointeur vers un pointeur.**

Exemple :

```
int tab[3] [4] =  {     {1,2,3,4 },
                        {5,6,7,8 },
                        {9,10,11,12}
                  } ;
```

- Le nom du tableau « **tab** » représente **l'adresse du premier bloc** qui a la valeur {1, 2, 3,4}.
- Une expression telle que **tab +1** correspond à l'adresse de tab, **augmentée de 4 entiers** (et non plus d'un seul !), ainsi **tab + 1** représente l'**adresse du deuxième bloc qui a la valeur {5, 6, 7,8}.**



Voici quelques exemples d'adresses équivalentes :

Type : int **	Type : int*	
tab	tab[0]	&tab[0][0]
tab+1	tab[1]	&tab[1][0]
tab+i	tab[i]	&tab[i][0]

- Les notations **tab et &tab[0][0]** correspondent toujours à **la même adresse** , mais **l'incrémentation de 1 n'a pas la même signification pour les deux.**
- **tab est un pointeur**, qui pointe vers un objet lui-même de type pointeur d'entiers (tab [0]).
- **tab[0]** représente l'adresse (pointeur) de début du premier bloc.
- **tab[1]** représente l'adresse de début du deuxième bloc.

Comment utiliser un pointeur de type int * pour parcourir un tableau à deux dimensions (int **) ?

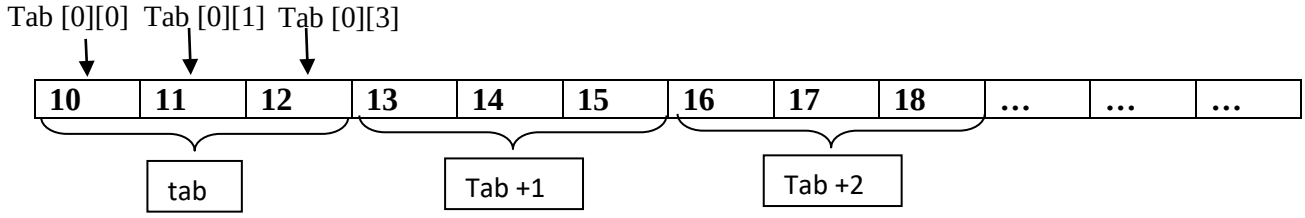
- Un **tableau 2 dimension en mémoire est un tableau unidimensionnel** dont chaque composante est un tableau unidimensionnel.

❖ **Exemple :**

- **Soit le tableau suivant « tab ».**

10	**11**	**12**
13	**14**	**15**
16	**17**	**18**

- **En mémoire le tableau est représenté comme suite :**



- **Pour utiliser un pointeur simple afin de parcourir le tableau à 2 dimensions :**

1. Convertir tab qui est de type int ** à int *.

Solution :

```
int tab[3] [3] ;
int  *P ;
P = (int *) tab ;
```

2. Trouver l'adresse de chaque élément en utilisant la formule suivante :

`&tab[i][j] = P+ i*TC +j`

i : indice de la ligne.

J : indice de la colonne.

TC : **T**aille réservée pour les **c**olonnes (nombre de colonnes déclaré)

P : Pointeur sur le début du tableau P= (int *) tab.

Exemple :

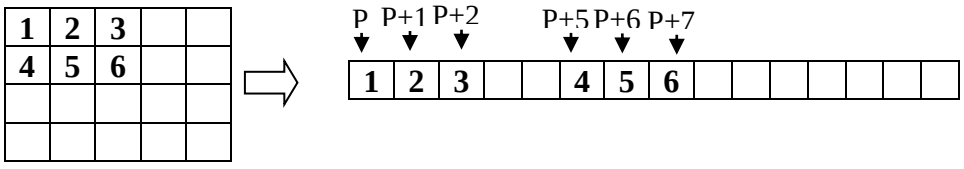
Si on déclare un tableau M de 4 lignes 5 colonnes.

```
int M [4][5] ;
```

TC = 5 ;

Attention : Si dans le programme nous avons utilisé uniquement 2 lignes, 3 colonnes, **TC** est = 5 (**réservé au début du programme).**

Exemple :



&M[i][j] = P+i*5+j

&M[**1**][**2**] = P+**1***5+**2** = P+ 7

&M[0][1] = P+0*5+1 = P+1

❖ **Exercice**

Soit le programme suivant :

```
int tab[3] [4] =  {      {10,20,30,40 },
                         {50,60, 70,80},
                         {90, 100, 110,120}
                  } ;
int * P ;
P=  (int*) tab ;
```

Quelles valeurs fournissent ces expressions ?

***P**	10
***(P+10)**	110
***P+5**	15
***tab[1]**	50
***(tab[2]+2)**	110

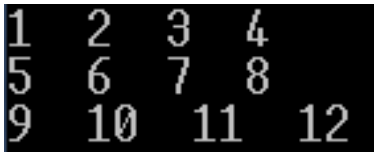
❖ **Exercice**

- Écrire un programme qui affiche le contenu d'un tableau 2D en utilisant les pointeurs.

Solution :

```
#include <stdio.h>
main()
{
int tab[3] [4] =  {      {1,2,3,4 },
                         {5, 6,7,8 },
                         {9,10,11,12}
                   };

int* P,i,j ;

P= (int*)tab ;

for(i=0; i<3 ; i++)
  {
    for(j=0; j<4; j++)
      {
        printf(" %d ", *(P+i*4+j));
      }
      printf("\n");
  }
}
```

❖ **Résultat :**



D'autres méthodes peuvent être utilisées pour parcourir un tableau en utilisant les pointeurs. Un exemple est présenté ci-dessous.

❖ **Méthode 2 :**

```
#include <stdio.h>

main()
{
int tab[3] [4] =  {      {1,2,3,4 },
                         {5,6,7,8 },
                         {9,10,11,12}
                  };

int * P,i,j ;


for (i=0; i<3 ; i++)
  {
    P= tab[i] ;
    for (j=0; j<4 ; j++, P++)
                {
                  printf("%d   ", *P);
                }
    printf("\n");
 }

 system("pause");
}
```

1.12 Tableaux et pointeurs - Résumé

Variable – valeur – Adresse	
int A	déclare une variable simple du type int.
A	désigne le contenu de A.
&A	désigne l'adresse de A.
Tableau simple et utilisation du nom du tableau comme pointeur	
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 le contenu de la composante i du tableau
Utilisation de pointeur avec un tableau simple	

int *P	déclare un pointeur sur des éléments du type **int**.
P	désigne l'adresse contenue dans P
***P**	désigne le contenu de l'adresse dans P
P=B	P pointe sur le tableau B
P	désigne l'adresse de la première composante
P+i	désigne l'adresse de la i-ème composante derrière P
***(P+i)**	désigne le contenu de la i-ème composante derrière P
Utilisation de pointeur avec un tableau 2 Dimensions	
int A[3][4]	Déclare un tableau 2 dimensions (3 lignes, 4 colonnes)
int *P = (int*)A	Déclarer un pointeur P et lui affecter l'adresse du tableau 2D. « Convertir A qui est de type int ** à int *)
&A[I][J]	P+I***4**+J « Formule (P+I***TC**+J) »
A[I][J]	*(P+I***4**+J)

1.13 Chaînes de caractères pointeurs.

- Une chaine de caractères est une suite d'octets en mémoire, terminée par un caractère nul ('\0').

'h'	'e'	'l'	'l'	'o'	'\0'

- Pour désigner une chaine de caractères, on donne simplement l'adresse de son premier octet.

❖ **Déclaration d'une chaine de caractère comme un tableau :**

```
char chaine[] = "hello";
char chaine[] = { 'h','e','l','l','o', '\0' };
char B[45] = "Deuxième chaîne un peu plus longue";
 char C[30];
```

Dans cette déclaration la chaine de caractères est un tableau (son nom est un pointeur constant) alors les instructions suivantes sont impossible:

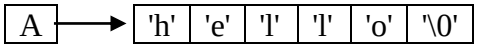
```
A = B;                    /* ERREUR !!! */
C = "Bonjour !";          /* ERREUR !!! */
Chaine = "Abdelhadi "     /* ERREUR !!! */
```

- Une instruction telle que chaine = "Salut"; **ne marche que lors de la définition et initialisation**.

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

❖ **Déclaration d'une chaine de caractère comme pointeur sur char:**

```
char *A = "Hello";
 char *B = "SMI S4";
char* salut ;
A = B;
salut = "coucou" ;
```



- A **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.

  A=B ;

  A= "Une autre chaine"
- **La chaîne constante** peut être lue, copiée ou affichée, **mais pas modifiée**.

❖ **Résumé :**

Une chaîne déclarée comme tableau :

Char A [20]="Bonjour" ;

- La taille du tableau est égale au nombre de caractères de la chaine +1 (' \0') .
- Le nom A doit toujours pointer sur la même adresse.
- Le contenu de A peut être changé.

Une chaîne déclarée comme pointeur :

Char *B = "Bonjour" ;

- B est un pointeur qui peut pointer sur n'importe quelle **chaîne de caractères constante** stockée en mémoire.
- **La chaîne constante** peut être lue, copiée ou affichée, **mais pas modifiée**.

❖ **Exercice 1**

- Ecrire une fonction qui retourne le miroir d'une chaîne de caractères.

  Exemple : Toto -- otoT
  SMI S4 -- 4S IMS

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

main()
{
        char mot[21], c;
        int i,j,n;

        printf("entrer un mot max.20 caractères : ");
        gets(mot);

        // claculer la taille de la chaine de caractère.
        n = strlen(mot);

        for(i=0,j=n-1;i<n/2;i++,j--)
        {
             c = mot[i];
             mot[i] = mot[j];
             mot[j] = c;

        }

         printf("le miroir est : %s \n", mot);

       system("pause");

}
```

❖ **Exercice 2**

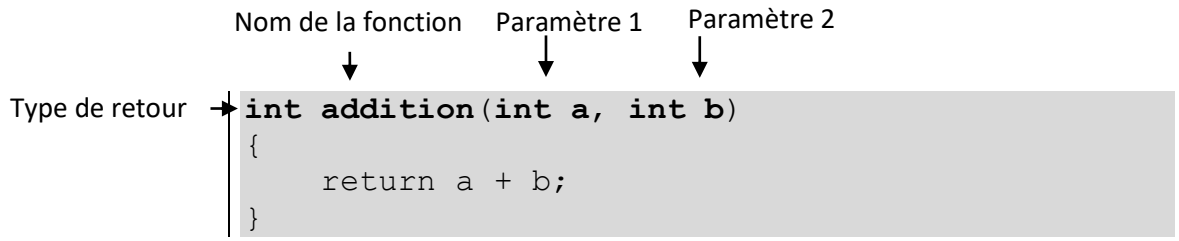
Ecrire un programme qui compte le nombre d'occurrences d'un caractère dans une chaîne de caractères.

```
#include <stdio.h>
#include <string.h>
main()
{
        char mot[21],c;
        char *ptr;
        int i,nbrOcc = 0;
```

```
        printf("entrer un mot max.20 caractères : ");
        scanf("%s", mot);
        printf("entrer un caractère : ");
        scanf(" %c", &c);

        for( ptr= mot; *ptr != '\0'; ptr++ )
        {
             if(*ptr == c ) nbrOcc ++;
        }

printf("le nombre d'occurrence de %c dans  %s est : %d  \n", c,
mot, nbrOcc );

}
```

1.14 Fonctions et pointeurs

❖ **Exemple d'une fonction :**



❖ **Le nom de la fonction** est **un pointeur constant** sur la première instruction de la fonction.

❖ **Déclaration d'un pointeur sur une fonction :**

```
Type de_retour (*nomPointeur) (types paramètres)
```

- L'exemple suivant présente la déclaration d'un pointeur sur une fonction qui **retourne un entier** et reçoit comme **paramètres deux entiers** :

```
int (*monPtrFonc) (int, int) ;
```

- Un autre exemple d'un pointeur sur une fonction qui ne retourne rien (procédure) avec trois paramètres.

```
void (*ptr_fonction) (int,char, float);
```

❖ **Affectation d'une fonction à un pointeur de fonction :**

- Déclaration de la fonction :

```
int addition(int a, int b)
{
    return a + b;
}
```

- Déclaration du pointeur sur la fonction :

```
  int (*ptr_Fonct) (int, int) ;
```

- Affectation :

```
ptr_Fonct = addition ;
```

OU Bien

```
ptr_Fonct = &addition ;
```

- Utilisation :

```
int n ;
n = ptr_Fonct (5,6) ;
```

ou bien

```
int n ;
n = (*ptr_Fonct) (5,6) ;
```

❖ **Confusion à éviter:**

- Ne pas confondre le pointeur sur fonction avec une fonction qui retourne un pointeur sur un type.

  Exemple : Une fonction qui retourne un pointeur sur le type «int».

```
 int * addition (int a, int b) ;
```

1.15 Allocation dynamique de la mémoire

- Il n'est pas toujours possible de savoir quelle quantité de mémoire sera utilisée par un programme.

- Par exemple, si vous demandez à l'utilisateur de vous fournir un tableau, vous devrez lui fixer une taille (**Allocation statique**), ce qui pose deux problèmes :
    - La taille ne convient peut-être pas à l'utilisateur qui a besoin de plus de mémoire; - manque de mémoire -
    - La taille fixée à l'avance est peut-être plus grande que le besoin de l'utilisateur. – gaspillage de mémoire –
- La solution sera alors de réserver la mémoire lors de l'exécution du programme. (**Allocation dynamique**).

❖ **La fonction malloc().**

❖ Syntaxe :

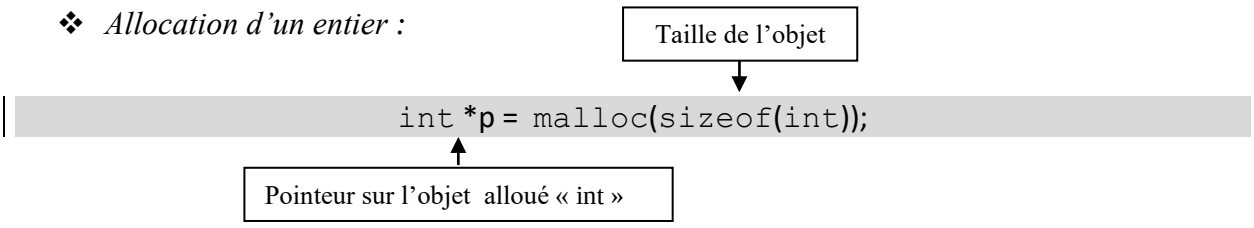
```
void *malloc(size_t taille);
```

❖ La fonction `malloc()` vous permet **d'allouer un objet de la taille fournie** en argument et **retourne l'adresse de cet objet** sous la forme d'un pointeur générique (peut être converti à n'importe quel autre type de pointeur).

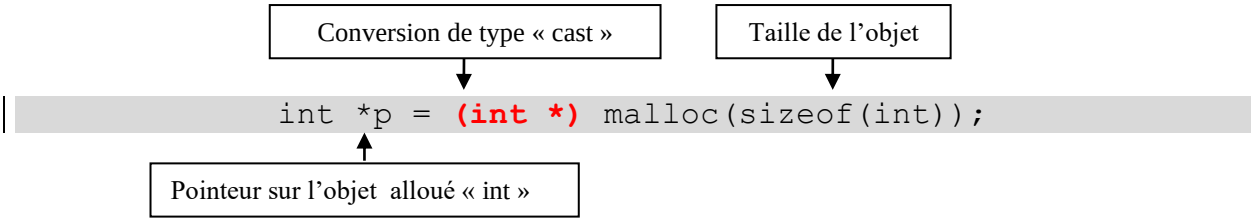
❖ En cas **d'échec** de l'allocation, elle retourne un **pointeur nul.**

Exemples :

❖ *Allocation d'un entier :*

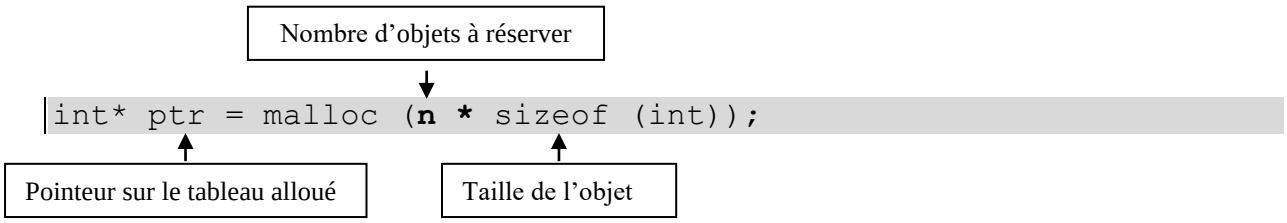


ou



Remarque* : caster la valeur de retour de malloc() **est inutile. En effet, malloc() renvoie un void *. Or, **en C, un pointeur void * est implicitement casté lors d'une affectation vers le type de la variable affectée.**

❖ *Allocation dynamique d'un tableau d'entiers de taille « n » :*



❖ *Allocation dynamique d'un tableau de float de taille « 10 » :*

```
float* ptr = malloc(10 * sizeof(float));
```

❖ *Allocation d'un tableau de 20 caractères:*

```
char *cptr =  malloc(20 * sizeof(char));
```

- **D'une manière plus générale :** pour allouer dynamiquement un objet de type *T*, il vous faut créer un pointeur sur le type *T* qui conservera son adresse.

Il est impératif de vérifier que le système a trouvé la quantité de mémoire souhaitée (la mémoire n'est pas infinie). Lorsque la tentative d'allocation échoue, la fonction malloc **renvoie un pointeur NULL**. Il faut donc tester si le pointeur retourné est égal à NULL; si c'est le cas, **on ne peut pas et on ne doit pas l'utiliser**:

```
float* ptr = malloc (1000 * sizeof (float));

if (ptr == NULL)
{
    printf ("l'allocation a échoué ") ;
    exit (1) ;
}
```

❖ **La fonction free().**

❖ Lorsqu'on n'a plus besoin d'un tableau alloué dynamiquement, il est impératif de libérer la mémoire.
❖ Cette opération se fait avec la fonction free().
❖ Syntaxe de free() : `void free (void *ptr) ;`

➢ **Remarque : malloc() et free()** sont déclarés dans le fichier en-tête **stdlib.h**

❖ **Utilisation des tableaux alloués :**

Les tableaux alloués de cette façon s'utilisent exactement comme les tableaux standards.

Exemple :

❖ Lire et afficher un tableau d'entiers alloués dynamiquement

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


main()
{
   int n, i, *P;

    printf("entrer le nombre d'éléments dans le tableau: ");
    scanf("%d",&n);

      // déclaration d'un tableau de N élément (allocation dynamique)
    int *tab =  malloc(n * sizeof(int));

      // Vérifier l'allocation

     if (tab == NULL)
         {
            printf("Echec de l'allocation\n");
            exit(1);
          }
          else
          {    // Remplissage comme un tableau simple
              for(i=0;i < n; i++)
              {
                  printf("donner l'élément %d  : ", i+1);
                  scanf("%d",&tab[i]);
                  printf("\n");
              }

              //affichage en utilisant un pointeur

              for(P=tab; P < tab + n ; P++)
              {
                  printf("L'élément %d ", (P-tab)+1);
                  printf("%d",*P);
                  printf("\n");
              }

             // Libération de la mémoire
              free(tab);
          }}
```

1.16 Avantages et inconvénients des pointeurs

Parmi les avantages des pointeurs :

- L’accès direct à la mémoire.
- La rapidité d’exécution des programmes.
- Les pointeurs fournissent un autre moyen de manipuler les tableaux.
- Les pointeurs sont utilisés pour l'allocation dynamique de mémoire.

- Les pointeurs permettent d'économiser beaucoup de mémoire.
- Les pointeurs sont utilisés pour construire différentes structures de données telles que des listes chainées, les files, les piles, etc.
- Etc.

Parmi les inconvénients des pointeurs :

- La notation du pointeur est difficile à lire.
- Les pointeurs non initialisés provoquent des erreurs.
- L'arithmétique des pointeurs doit être faite avec précaution.
- Le bloc alloué dynamiquement doit être libéré explicitement.
- Etc.

2) Chapitre II : Structures de données.

2.1 Définition d'une structure.

- ❖ Une structure est un ensemble de variables de types éventuellement différents, regroupés sous un même nom.

- ❖ La structure permet de créer un type de données qui peut être utilisé pour regrouper des éléments de types éventuellement différents.

- ❖ Exemple :
  - structure voiture (marque, nbr de cylindre, carburant, couleur, prix).
  - Structure produit(désignation, prix, quantité, date_expiration).

Structure **élève** (nom, prénom, âge, moyenBac, codeMassar).

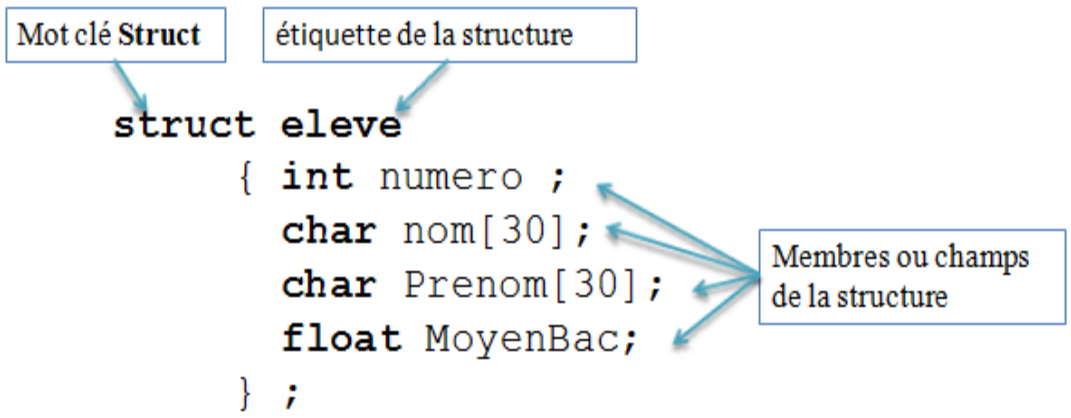
« La Structure est comme une boîte qui regroupe plusieurs données différentes. »

2.2 Déclaration d'une structure

● **Exemple 1:**

```
struct produit
               {
                 int numero ;
                 int qte ;
                 float prix ;
              };
```

● **Exemple 2:**



● **Remarque:** Cette déclaration définit un modèle de structure, mais ne réserve pas de variables correspondant à cette structure.

2.3 Définition d'une variable de type structure

● Méthode 1:

Exemples :

```
struct eleve   ahmed;
struct produit ordinateur;
struct voiture renault;
```

```
struct Point
{
     int x, y;
};

int main()
{
     struct Point p1;
}
```

● Méthode 2: déclaration de la variable avec la déclaration de structure

Exemple :

```
struct eleve
     {
       int numero ;
       char nom[30];
       char Prenom[30];
       float MoyenBac;
     } ahmed;
```

```
struct Point
{
     int x, y;
} p1, p2;
```

- **Exercice :**

  Déclarer une structure voiture avec les champs suivants : marque modèle, puissance_Fiscale et prix;
  Déclarer 2 variables de types voitures.

2.4 Initialiser les membres de la structure

● **Erreur à éviter:**

```
struct Point
{
int x = 0; // ERREUR COMPILATEUR: impossible d'initialiser les membres ici
int y = 0; // ERREUR COMPILATEUR: impossible d'initialiser les membres ici
};
```

● **Initialisation séquentielle**

L'initialisation séquentielle permet de spécifier une valeur pour un ou plusieurs membres de la structure en suivant l'ordre de la définition.

```
struct eleve

     {  int   numero ;
        char  nom[30];
        char  Prenom[30];
        float MoyenBac;
     };
```

```
struct eleve e1= { 123, "Ahmed", "Ali", 12.5 };
```

● **Initialisation sélective**

Il est possible spécifier les champs à initialiser.

```
struct eleve
     {
        int   numero ;
        char  nom[30];
          char  Prenom[30];
        float MoyenBac;
     };
```

```
struct eleve e1= {.nom = "Ahamed",  .moyen =12,5 };
```

2.5 Accès à un membre de la structure

L'accès à un membre d'une structure se réalise à l'aide de la variable de type structure et de l'opérateur • suivi du nom du champ visé.

```
variable.membre
```

● **Exemples:**

```
struct eleve  e1 ;

// Pour accéder aux membres de e1

e1.nom;
e1.moyenBac ;
```

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

struct eleve
     {  int   numero ;
        char  nom[30];
        char  prenom[30];
        float moyenBac;
     };


main()
{
       struct eleve e1 = {123, "Ahmed", "Ali", 12.5 };

       printf("Le num : %d \n", e1.numero);
       printf("Le nom : %s \n", e1.nom);
       printf("Le prénom : %s \n", e1.prenom);
       printf("La moyenne du Bac %f: \n", e1.moyenBac);

       system("pause");
}
```

2.6 Tableau de structures

Il est possible de créer un tableau de structures de la même façon qu'un tableau d'entiers, de réels de caractères, etc.

Syntaxe :

```
struct etiquette  tab[n];
```

● Exemple:

L'exemple suivant présente la déclaration, le remplissage et l'affichage d'un tableau de structures.

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

struct Eleve
    {  int   numero ;
       char  nom[30];
       char  prenom[30];
       float moyenBac;
    };

main()
{

      // tableau de structures
      struct Eleve tab[10];
      int i;

      //Accéder aux membres du tableau
    for(i=0;i<10;i++)
    {
      printf("Entrer les informations de l'Eleve: %d \n", i+1);
      printf("Le numero :  \n");
      scanf("%d",&tab[i].numero);
      printf("Le nom : \n");
      scanf("%s",&tab[i].nom);
      printf("Le prénom : \n");
      scanf("%s",&tab[i].prenom);
      printf("La Moyenne : \n");
      scanf("%f",&tab[i].moyenBac);
      }

       for(i=0;i<10;i++)
     {
       printf("les informations de l'Eleve: %d \n", i+1);
       printf("Le num : %d \n", tab[i].numero);
       printf("Le nom : %s \n", tab[i].nom);
       printf("Le prénom : %s \n", tab[i].prenom);
       printf("La moyenne du Bac %f: \n", tab[i].moyenBac);
       }

      system("pause");
 }
```

2.7 Typedef et structures

Pour créer une variable de type structure vous êtes obligé de suivre la syntaxe suivante :

struct étiquette variable.

Par exemple :

- **struct eleve** ahmed ;
- **struct voiture** renault ;
- **struct produit** p1.

Pour ne pas répéter le mot clé **strcut** à chaque déclaration de variable de type structure vous pouvez utiliser ***typedef***.

● Exemples:

Sans utilisation de Typedef	Avec Typedef

struct personne
{
int numero ;
char nom[30];
char Prenom[30];
 	
};
 	
struct personne E1, E2 ;
	typedef struct
	{
	int numero ;
	char nom[30];
	char Prenom[30];
	float MoyenBac;
	}**s_eleve;**
	
	s_eleve E1, E2 ;
	
	L'écriture **s_ *eleve*** n'est plus possible dans le cas d'une **structure récursive** (c'est-à-dire un type structuré dont un des champs est du même type structuré), car le compilateur doit connaître tous les mots qu'il rencontre pendant sa lecture linéaire du fichier à compiler.
	Voici un exemple de définition de structure récursive ***s_eleve***:
	
	typedef struct eleve
	{
	int numero ;
	char nom[30];
	char Prenom[30];
	float MoyenBac;
	struct eleve * frere **;**
	
	}**s_eleve**;
	
	s_eleve E1, E2 ;

2.8 Pointeurs et structures

Comme les types primitifs, nous pouvons avoir un pointeur sur une structure. Si nous avons un pointeur sur la structure, les membres sont accessibles en utilisant l'opérateur flèche (→).

***Exemple* :**

```
#include<stdio.h>

Typedef struct
{
int   numero ;
char  nom[30];
char  Prenom[30];
float MoyenBac;
}eleve;

main()
{
    eleve E1 = {123,"Ahmed","Ben",14.5};
    eleve * P ;
    P=&E1 ;

    Printf("Le nom :  %s \n le prénom : %s", P→nom, P→prenom) ;
}
```

❖ **Remarque :**

E1.nom	est équivalente à	**P→nom**
&E1.nom	est équivalente à	**&P→nom**

2.9 Allocation dynamique de structures.

❖ Pour **l'allocation dynamique d'une variable** de type structure:

```
typedef struct
{
    char nom[20];
    float moyenne;
} etudiant;

main()
{
     // Allocation dynamique d'une variable
  etudiant * PtrE1 =  malloc(sizeof(etudiant));

 if (PtrE1 ==NULL)
    {
        printf("\n Allocation dynamique impossible !");
        exit(1) ; // on quitte le programme
    }  }
```

❖ Pour **l'allocation dynamique d'un tableau de structures:**

```
typedef struct
{
    char nom[20];
    float moyenne;
} etudiant;

main()
{
    int n =10;

// Allocation dynamique d'un tableau de structures de taille n
  etudiant * PtrE1 =  malloc( n * sizeof(etudiant));

 if (PtrE1 ==NULL)
    {
        printf("\n Allocation dynamique impossible !");
        exit(1) ; // on quitte le programme
    }
}
```

❖ **Exercice :**

Soit la structure suivante : étudiant (nom, age, moyen);

- ❖ Écrire un programme qui lit et affiche un tableau de structures étudiant de taille n (déterminée par l'utilisateur).

❖ **Solution :**

```
#include <stdio.h>
typedef struct etudiant
{
     char nom[20];
     float moyenne ;
     int age ;
}   etudiant;
```

```c
main()
{

   etudiant *tab_etud;
   int i,n;

    printf("Tapez le nombre d'etudiants  :");
    scanf("%d", &n );

    tab_etud = (etudiant *)malloc( n* sizeof(etudiant));


    if (tab_etud==NULL)
    {
        printf("\n Allocation dynamique impossible !");
        exit(1) ; // on quitte le programme
    }


   /* Remplissage du tableau */
   for (i=0 ; i<n ; i++)
   {
      printf("Entrer le nom de l'etudaint %d  : \n", i+1);
      scanf("%s", tab_etud[i].nom);
      printf("Entrer la moyenne de l'etudaint %d  : \n", i+1);
      scanf("%f",&tab_etud[i].moyenne);
      printf("Entrer l'age de l'etudaint %d  : \n", i+1);
      scanf("%d", &tab_etud[i].age);
   }


       /* Affichage du tableau  en utilisant un pointeur*/

    etudiant *p ;
    for ( p=tab_etud; p<tab_etud+n ; p++)
    {
       printf("..................................\n");
       printf("L'etudaint %d  : \n", (p-tab_etud)+1);
       printf("Nom    : %s ", p->nom);
       printf("Moyenne    : %f ", p->moyenne);
       printf("L'age    : %d n", p->age);
    }
    /* LIBERATION MEMOIRE : */
    free(tab_etud);
    system("pause");
}
```

2.10 Les structures et les fonctions.

● **Passage d'une variable de structure en tant qu'argument d'une fonction : *Passage par valeur.***

```c
#include < stdio.h>

struct etudiant{
    char prenom[20];
    int age;
};
void afficher(struct etudiant et)
{
    printf(" prenom : %s \n", et.prenom);
    printf(" age : %d \n", et.age);
}

int main(void)
{
    struct etudiant et1 = {"Mostafa", 24};

    afficher(et1);

    return 0;
}
```

● **Passage d'une variable de structure en tant qu'argument d'une fonction :** ***Passage par référence (adresse)***

```c
#include < stdio.h>

struct complex{
    float R; // partie réelle
    float I; // Partie Imaginaire
};
void Ajouter(struct complex c1, struct complex c2, struct complex *res)
{
    res->R=(c1.R + c2.R);
    res->I=(c1.I + c2.I);
}

int main(void)
{
    struct complex c1 = {2.5, 3}, c2={1.24,4}, somme;

    Ajouter(c1, c2, &somme);

    printf("res.R = %f  et res.I = %f", somme.R,somme.I);

    return 0;
}
```

● **Passage d'un tableau de structure en tant qu'argument**

- ***Méthode 1*** :

```c
void myFunction(struct  etudiant  *t , int n )
{
. . .
}
```

- ***Méthode 2 :***

```
void myFunction(struct  etudiant t[100] )
{
. . .
}
```

- Méthode 3 :

```
void myFunction(struct  etudiant t[], int n )
{
. . .
}
```

● **Renvoyer une structure à partir d'une fonction**

- **Exemple 1 :**

```
struct etudiant{
     char nom[20];
     int age;
};

struct etudiant Saisir(){
     // variable local;
     struct etudiant e;

     printf("Saisir le prénom : ");
     scanf("%s",&e.nom);

     printf("saisir l'age : ");
     scanf("%d",&e.age);

     // retourner une copie de e;
     return e;
}

int main(void)
{
     struct etudiant et;
     et=Saisir();

     printf("voici vos infos : ");
     printf("Prénom : %s",et.nom);
     printf("age : %d",et.age);

     return 0;
```

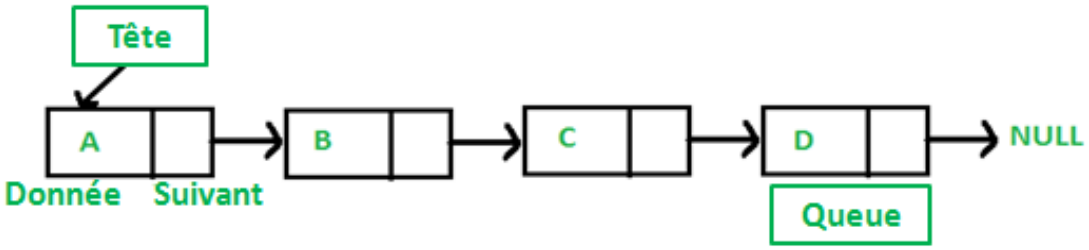
- **Exemple 2**

```c
struct etudiant{
    char prenom[20];
    int age;
};

struct etudiant *Saisir(struct etudiant *e){

    printf("Saisir le prénom : ");
    scanf("%s",&e->prenom);

    printf("saisir l'age : ");
    scanf("%d",&e->age);

    return e;
}

int main(void)
{
    struct etudiant et;
    struct etudiant *et2;
    et2=Saisir(&et);

    printf("voici vos infos : \n");
    printf("Prénom : %s \n",et2->prenom);
    printf("age : %d",et2->age);

    return 0;
}
```

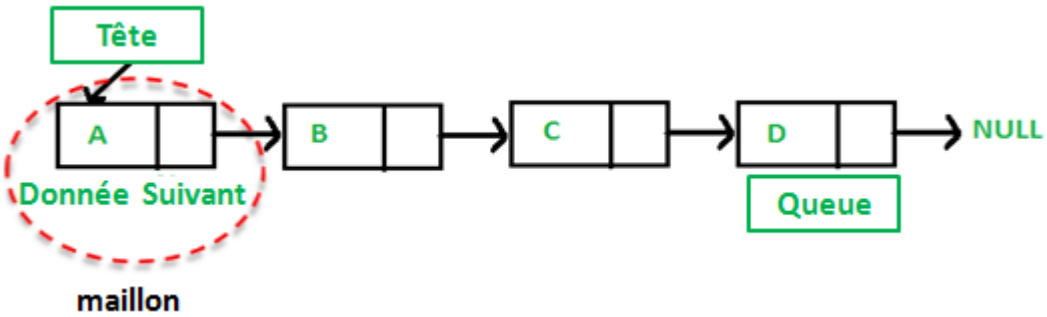
3) CHAPITRE 3 : Les listes chainées, les piles et les fils.

3.1 Définition :

- Une liste chaînée est une structure de données linéaire, dans laquelle les éléments ne sont pas stockés à des emplacements de mémoire contigus.
- Les éléments d'une liste chaînée sont liés à l'aide de pointeurs comme indiqué dans l'image ci-dessous:



- L'élément de base d'une liste chaînée s'appelle le maillon (structure). Il est constitué : d'un ou plusieurs champs de données et d'un pointeur vers le maillon suivant. S'il n'y a pas de maillon suivant, le pointeur vaut NULL.



3.2 Listes chainées VS. Tableaux.

- Dans un tableau les données sont stockées à des emplacements de mémoire contigus. Contrairement à une liste chainée.
- Contrairement au tableau, la liste n'est pas allouée en une seule fois, mais chaque élément est alloué indépendamment (maillon par maillon).
- La liste chainée offre une facilité d'insertion / suppression surtout en cas d'éléments triés. Contrairement au tableau où nous devons insérer et déplacer le reste des éléments.
    - Exemple :
      Soit le tableau suivant id [] = [1000, 1010, 1050, 2000, 2040].

Pour insérer la valeur 1005 et maintenir l'ordre trié, nous devons déplacer tous les éléments après 1000 (sauf 1000).

- L'accès aléatoire n'est pas autorisé dans une liste chainée. Nous devons accéder séquentiellement aux éléments à partir du premier nœud.
- Dans une liste chainée, un espace mémoire supplémentaire pour un pointeur sur l'élément suivant est requis avec chaque élément de la liste.

3.3 Implémentation de la liste chainée.

- La structure de données de la liste chainée contient deux parties :

  - la valeur que vous voulez stocker,

  - L'adresse de l'élément suivant.

Exemple :

```
typedef  struct Maillon
{
  int donnee;
  struct Maillon* suivant;

} Maillon;
```

3.4 Fonctions de gestion de liste

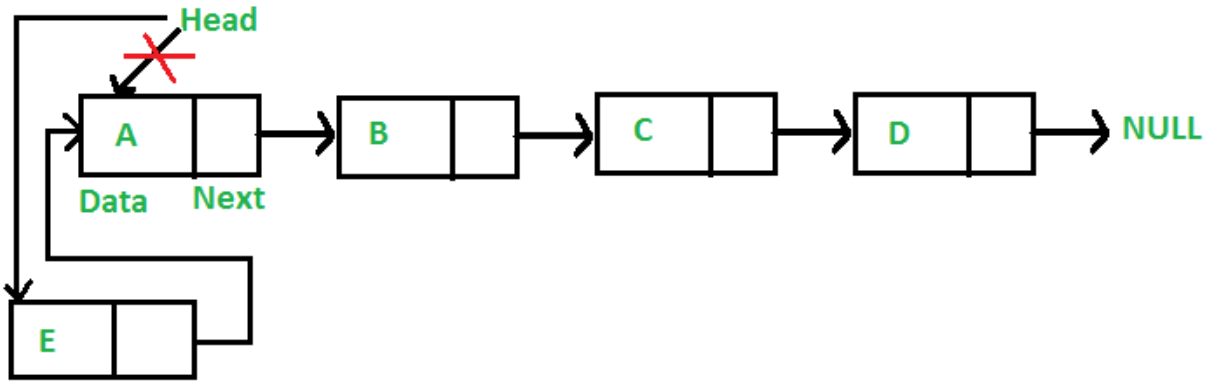
- **Remarque:**

La liste est connue par l'adresse du premier élément si cette adresse n'est pas mémorisée la tête de la liste disparait, donc il faut toujours avoir la tête de la liste mémorisée.

3.4.1 Ajout d'éléments au début de la liste

- Dans cette fonction on veut insérer un élément au début de la liste.

1) Déclarer une fonction qui prend comme paramètres la tête de la liste et la nouvelle valeur à insérer, et retourne la nouvelle tête de la liste.
2) Créer le nouvel élément en utilisant l'allocation dynamique.
3) Affecter la valeur reçue en paramètre au nouvel élément crée.
4) Comme on veut que ce nouvel élément soit en premier de cette liste, on fait le chainage vers la tête par l'instruction **`NouvelElement→suivant = tete.`**

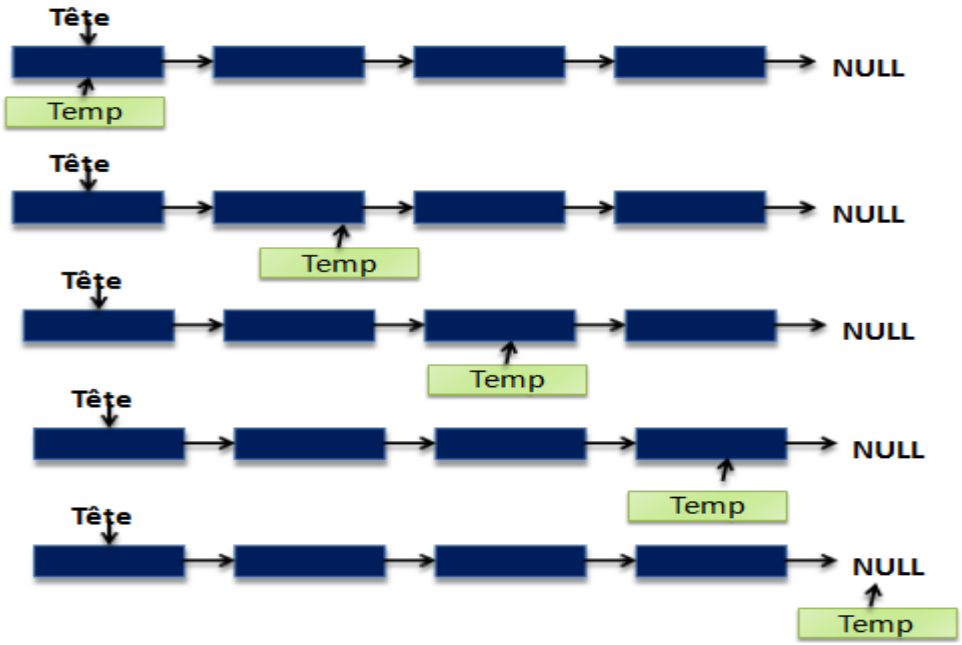



```
Maillon * ajouterEnTete(Maillon * tete, int valeur)
{
    /* créer un nouvel élément */
    Maillon * nouvelElement = malloc(sizeof(Maillon));
    if (nouvelElement == NULL)
    {         printf("Allocation échouée");
              exit(1);
    }

  /* on assigne la valeur au nouvel élément */
    nouvelElement->donnee = valeur;

    /* on assigne l'adresse de l'élément suivant au nouvel
élément */
    nouvelElement->suivant = tete;

    /* on retourne la nouvelle liste, c.-à-d. le pointeur sur le
premier élément */
    return nouvelElement;
}
```

3.5 Affichage d'éléments de la liste.

❖ Pour afficher les éléments de la liste :

1) Créer une variable temporaire qui pointe sur la tête.
2) Passer d'un élément à un autre en utilisant temporaire = temporaire ->**suivant.**



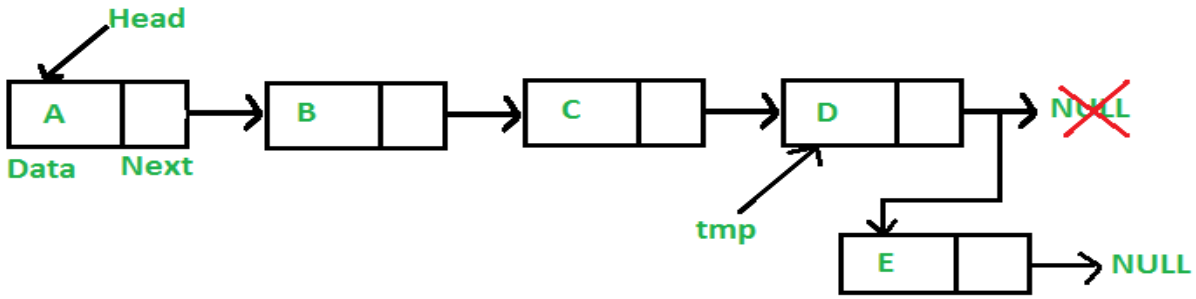

```
void AfficheListe(Maillon* tete)
{
    /* créer un Maillon temporaire qui pointe sur le début de la liste */
     struct Maillon *temp = tete;

    /* vérifier si c'est pas null */
     while (temp != NULL)
    {
      printf(" %d ", temp->donnee);
     /* pointer sur le maillon suivant */
      temp = temp->suivant;
     }
}
```

3.6 Rechercher un élément dans la liste.

```
bool RechercheListe(struct Maillon* tete, int n )
   {
       struct Maillon *temp = tete;

       while (temp != NULL)
       {

       if (temp->donnee == n) {return true; }

       temp = temp->suivant;
       }

  return false;
  }
```

3.7 Ajouter un élément à la fin de la liste.

❖ Pour ajouter un élément à la fin de la liste il faut parcourir la liste jusqu'au dernier élément qui pointe sur NULL. Puis pointer le suivant du dernier élément sur le nouvel élément créé.

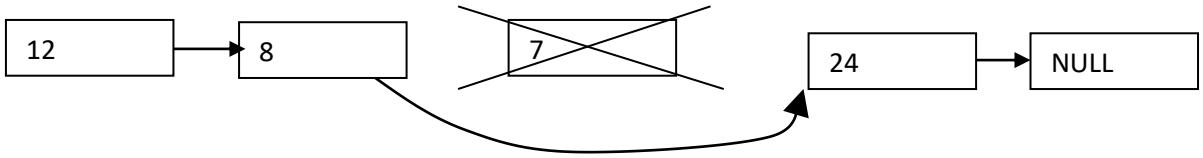



```
Maillon * ajouterEnFin(Maillon * tete, int valeur)
{
    /* On crée un nouvel élément */
    Maillon* nouvelElement = malloc(sizeof(Maillon));
    if (nouvelElement == NULL)
    {         printf("Allocation échouée");
              exit(1);
    }
    /* On assigne la valeur au nouvel élément */
    nouvelElement->donnee= valeur;
    /* On ajoute en fin, donc aucun élément ne va suivre */
    nouvelElement->suivant = NULL;

     if(tete == NULL)
    {
     /* Si la liste est vide il suffit de renvoyer l'élément créé */
        return nouvelElement;
    }
    else
    {   /* sinon, on parcourt la liste à l'aide d'un pointeur temporaire et on
indique que le dernier élément de la liste est relié au nouvel élément */

        Maillon* temp=tete;
        while(temp->suivant != NULL)
        {
            temp = temp->suivant;
        }
        temp->suivant = nouvelElement;
        return tete;
    }
}
```

3.8 Supprimer un élément de la liste.

- Pour supprimer un élément, il faut premièrement le chercher, ensuite selon sa position dans la liste réaliser le traitement convenable.
    - Si l'élément à supprimer est le premier dans la liste (la tête), il faut un pointeur sur le deuxième élément **(nextElement)** pour le retourner ensuite, car il deviendra la nouvelle tête de la liste. Puis supprimer l'élément (le libérer avec la fonction free).
    - Si l'élément n'est pas le premier, il faut parcourir la liste avec deux pointeurs ; «Pour ne pas créer un autre pointeur temporaire, en utilisera **(nextElement**) pour pointer sur l'élément courant ; et **(precedent)** pour pointer sur l'élément précédent. Une fois l'élément est trouvé, il faut concaténer le précédent avec le suivant de l'élément courant (à supprimer).

precedent →suivant = nextElement→suivant.




```
Maillon * supprimerElement(Maillon * tete, int valeur)
{
       /* Si la liste est vide : ne rien faire */
        if (tete == NULL ) { return NULL;}

      /* Creer un pointeur sur le premier élément */
        Maillon *precedent = tete;
       /* Creer un pointeur sur le deuxième élément */
       Maillon *nextElement = tete->suivant;

       /* Si la valeur est en tête de la liste */
       if(tete->donnee == valeur )
       {
              free(tete);
                return nextElement;
       }

       /* Si la valeur N'est pas en tête de la liste */
        while (nextElement != NULL)
```

```
        {
              if(nextElement->donnee == valeur)
              {
                /* concatener le précédent et le suivant */
                   precedent->suivant =nextElement->suivant;
                   free(nextElement);
                   return tete;
              }

/* On déplace nextElement (mais precedent garde l'ancienne valeur de
NextElement */
           precedent = nextElement;
          nextElement = nextElement-> suivant ;
        }

        return tete;
    }
```

3.9 Exemple de programme principal

La liste est connue par l'adresse du premier élément si cette adresse n'est pas mémorisée la tête de la liste disparait, donc il faut toujours avoir la tête de la liste mémorisée. C'est pourquoi il suiffait de déclarer un pointeur de type structure utilisé dans la liste (Maillon) et l'initialiser par NULL.

```
Struct Maillon * tete = NULL;
```

Ensuite il ne reste que l'appel aux différentes fonctions précédemment déclarées.

```
int main()
{
    /* déclaration d'un pointeur sur le début de la liste*/
    Maillon * Tete = NULL;
          /* Ajouter au début */
    Tete =  ajouterEnTete(Tete,15);
    Tete =  ajouterEnTete(Tete,14);
          /* Ajouter à la fin */
    Tete = ajouterEnFin(Tete,25);
          /* Supprimer un élément */
    Tete = supprimerElement(Tete,15);
          /*Afficher la liste */
    AfficheListe(Tete);
}
```

3.10 Les piles

Les piles et les files sont des exemples de listes chainés.

● Les Piles.

- Dans une pile (stack en anglais): **Le dernier** élément qui a été ajouté **est le premier à sortir** « **LIFO : Last In First Out** ». Prenons un exemple : une pile d'assiettes à laver. On les met les unes sur les autres et pour les laver, on prend celle du dessus. Un deuxième exemple : une pile de dossiers sur un bureau. On les met les uns au-dessus des autres et on retire le premier, donc le dernier arrivé. Les piles répondent à ce principe : *dernier arrivé, premier servi*.

Pile d'assiettes

- L'opération d'ajout dans une pile s'appelle **: « Empiler ».**

- L'opération d'enlever un élément de la liste s'appelle **: « Dépiler ».**

- Le premier élément dans la pile est nommé **« sommet ».**

De nombreuses applications s'appuient sur l'utilisation d'une pile, on peut citer :

- Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton « Afficher la page précédente ».
- La fonction « Annuler la frappe » (en anglais « Undo ») d'un traitement de texte mémorise les modifications apportées au texte dans une pile.
- Vérification de parenthèse d'une chaine de caractères ;
- La récursivité (une fonction qui fait appel à elle-même) ;
- **Etc.**

●Les files

- Dans une file (queue en anglais) : **Le premier élément** qui a été ajouté **est le premier à sortir** « **FIFO : First In First Out** ».Un exemple tout simple : une file d'attente. Le premier arrivé est le premier à sortir (à être servi).

File d'attente.

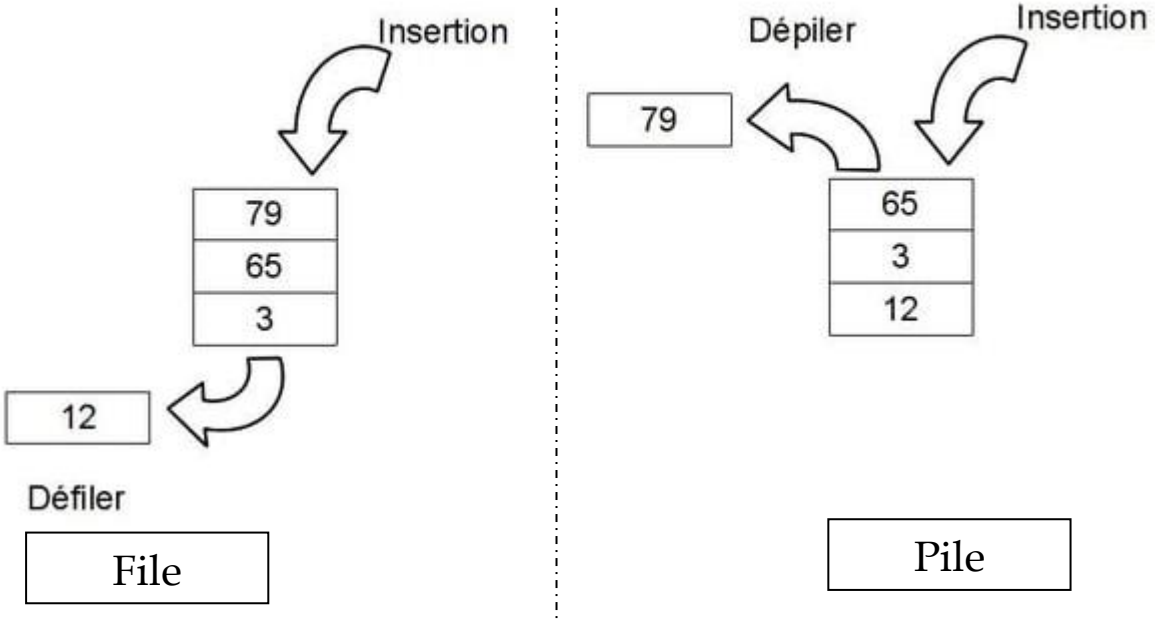
- Ajouter un nouvel élément à la file est appelé **Enfiler**.

- Enlever un élément de la file est appelé **Défiler.**

De nombreuses applications s'appuient sur l'utilisation d'une pile, on peut citer :

- Les serveurs d'impression, qui doivent traiter les requêtes dans l'ordre dans lequel elles arrivent, et les insèrent dans une file d'attente (ou une queue) ;
- Certains systèmes d'exploitation, qui doivent exécuter plusieurs tâches dans le même ordre d'arrivées.

●**Comparaison entre la Pile et la File.**



- **Dans une pile on ajoute à la tète de la pile** « sommet » **et on enlève à partir de la tête,** ce qui fait que l'élément ajouté en dernier lieu est le premier à récupérer.
- **Dans une liste on ajoute à la queue et on enlève à partir de la tête**, ce qui fait que le premier élément ajouté est le premier à récupérer.

●Implémentation des piles et des files

- Il y **a deux méthodes pour créer des piles et des files**:
  - Méthode 1 : Utilisation des tableaux (piles et files statiques).
  - Méthode 2 : Utilisation des listes chainées (piles et files dynamiques).

- Dans ce cours on s'intéresse aux piles et files dynamiques (Utilisation des listes chainées).

❖ ***Pour créer une pile*** : créer une liste chainée ; ajouter les éléments à la tête de la liste « Empiler » et récupérer les éléments à partir la tête (sommet) « Dépiler ».

❖ ***Pour créer une file*** : créer une liste chainée ; ajouter les éléments à la fin de la liste « Enfiler » et récupérer les éléments à partir de la tête « Défiler».

❖ **Exemple de gestion d'une pile :**

- **La déclaration de l'élément de base dans la pile**

```
typedef struct Maillon
{
    int donnee;
    struct Maillon* suivant;

}Maillon;
```

- **La déclaration la fonction empiler (ajouter au début).**

```
Maillon * empiler(Maillon * sommet, int valeur)
{
    /* On crée un nouvel élément */
  Maillon * nouvelElement = malloc(sizeof(Maillon));
  if (nouvelElement == NULL)
    {         printf("Allocation échouée");
              exit(1);
    }

  nouvelElement->donnee = valeur;
```

```
   /* On ajoute le nouvel élément à la tête de la pile */

   nouvelElement->suivant = sommet;

    /* On retourne le pointeur sur le premier élément */
    return nouvelElement;
}
```

- **La déclaration la fonction dépiler (retirer un élément).**

Nous envoyons à la fonction dépiler le sommet de la pile (*passage par référence d'un pointeur*, on cherche à changer le sommet et non la valeur sur laquelle il pointe, d'où l'utilisation de Maillon ** sommet). Lors de l'appel de cette fonction, il faut envoyer l'adresse du pointeur. La fonction d'épiler retourne la valeur dépilée ou -1 si la pile est vide.

```
int depiler(Maillon **sommet)
{
       /* Si la liste est vide :  Returner -1 */
       if (*sommet == NULL ) {  return -1;
                              }

      /* Créer un pointeur   sur le sommet */
        Maillon *temp = *sommet;

        /* Modifier le sommet */
        *sommet = temp->suivant;

      /* Créer une copie des valeurs à dépiler */
       int valeur = temp->donnee;

      /* Libérer le sommet */
        free(temp);

      /* retourner le nouveau sommet */
        return valeur;
}
```

- **La déclaration de la fonction d'affichage**

```
void afficherPile(Maillon* sommet)
{
  /* creer un Maillon temporaire qui pointe sur le debut de la liste */
    struct Maillon *temp = sommet;

    /* verifier si c'est pas null */
    while (temp != NULL)
```

```
    {
      printf(" %d ", temp->donnee);
      /* pointer sur le maillon suivant */
      temp = temp->suivant;
     }
}
```

- **Exemple de programme principal.**

```
int main()
{
  /* déclaration d'un pointeur sur le sommet de la pile*/
    Maillon * sommet = NULL;
  /* Empiler */
    sommet =  empiler(sommet,10);
    sommet =  empiler(sommet,20);
    sommet =  empiler(sommet,30);
    sommet =  empiler(sommet,40);
    sommet =  empiler(sommet,50);



    // Afficher
    printf("Affichage de la pile :");
    afficherPile(sommet);

     // Dépiler
     int n =   depiler(&sommet);
      // Afficher
    printf("\nAffichage de la pile: ");
    afficherPile(sommet);

}
```

❖ **La gestion d'une file.**

- La gestion d'une file c'est une gestion d'une liste chainée simple :
  - L'ajout c'est à la fin de la liste (voir listes chainées).
  - L'extraction c'est à partir de la tête de la liste (comme pour la pile).

Ce cours s'inspire de plusieurs cours disponibles sur le web :

Références

- Programmer en langage C Claude Delannoy
- INTRODUCTION A LA PROGRAMMATION EN ANSI-C https://www.ltam.lu/cours-c/
- Programmation en langage C, Anne Canteaut https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/
- https://www.tutorialspoint.com/cprogramming/
- https://zestedesavoir.com/tutoriels/755/le-langage-c-1/
- https://www.labri.fr/perso/billaud/travaux/Pointeurs/pointeurs.html#retour-sur-les-expressions-indic%C3%A9es
- https://fr.wikibooks.org/wiki/Programmation_C/Pointeurs
- http://public.iutenligne.net/informatique/algorithme-et-programmation/priou/LangageC/index.html
- https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c
- http://www.msc.univ-paris-diderot.fr/~daerr/teaching/phynumM1/c.html
- https://www.levenez.com/lang/c/faq/fclc012.html
- https://www.geeksforgeeks.org/data-structures/