❖ **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**

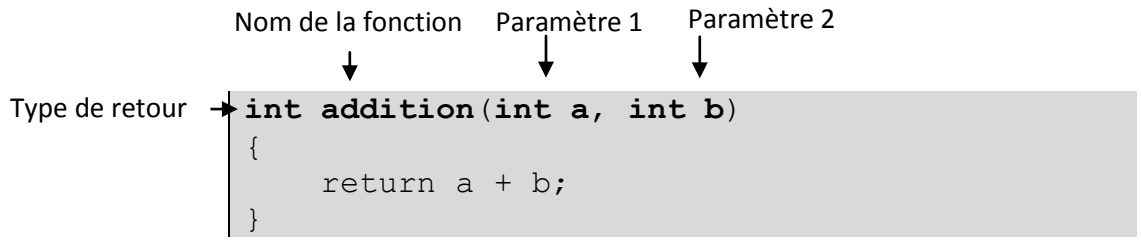
Ecrire un programme qui compte le nombre d'occurrences d'un caractère dans une chaîne de caractères.

```
#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().**

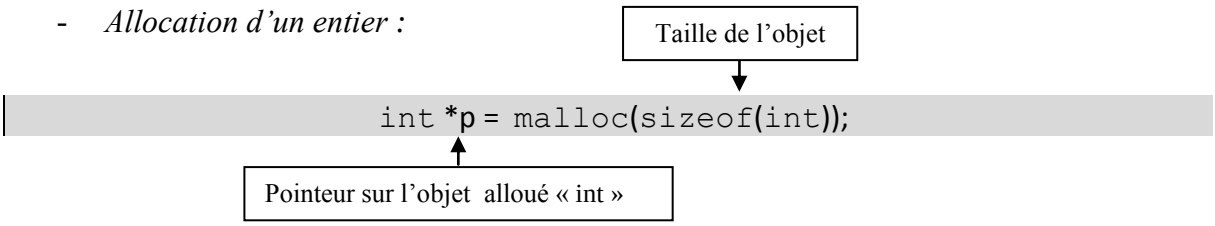
- Syntaxe :

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

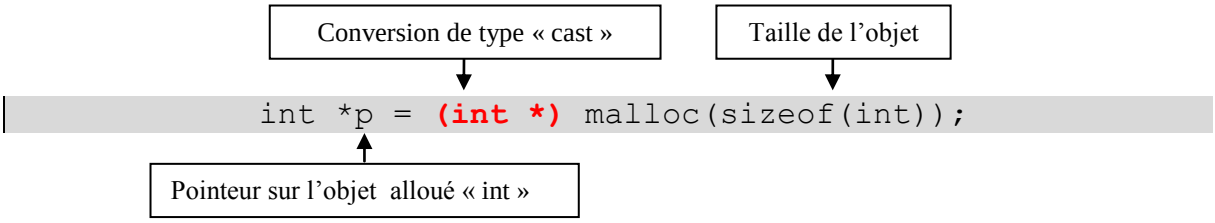
- La fonction `malloc()` vous permet **d'allouer un objet de la taille fournie** en argument et **retourne l'adresse de cet objet** sous la forme d'un pointeur générique (peut être converti à n'importe quel autre type de pointeur).
- En cas **d'échec** de l'allocation, elle retourne un **pointeur nul.**

Exemples :

- *Allocation d'un entier :*

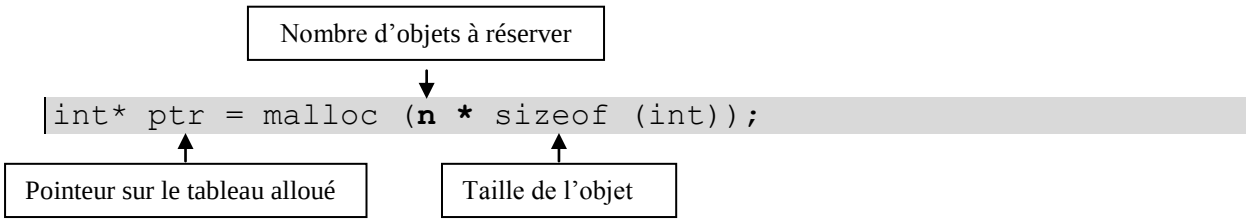


ou



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 et types abstraits.

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).

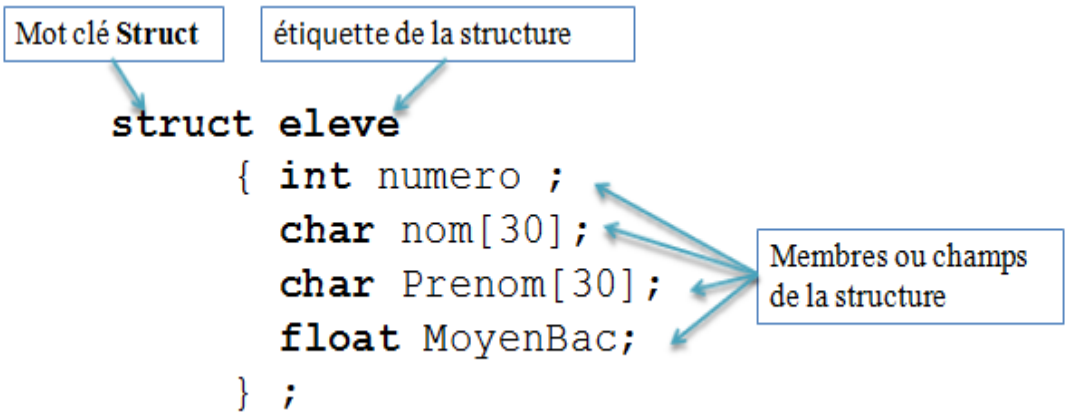
« La Structure est comme une boîte qui regroupe plusieurs données différentes. »

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` `varaiable`.

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) ;
}
```

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");
}
```