Université Sidi Mohamed Ben Abdallah de Fès
Faculté Polydisciplinaire Taza

Département : Mathématiques, Physique et Informatique

Filière : Sciences Mathématiques et Informatique

Semestre : S4

Module: Programmation 2 (Langage C)

Prof, A.EL AFFAR

Année universitaire 2020- 2021

PLAN

1. Rappel général
2. Pointeurs en langage C
3. Les pointeurs et tableaux (Allocation dynamique).
4. Les fonctions et pointeurs.
5. Les types de variables complexes(enregistrements, unions, énumérés)
6. Gestion des fichiers.

1. Rappel général

Deux sortes de programmes

Le système d'exploitation : ensemble des programmes qui gèrent les ressources matérielles et logicielles ; il propose une aide au dialogue entre l'utilisateur et l'ordinateur : l'interface textuelle (interprète de commande) ou graphique (gestionnaire de fenêtres). Il est souvent multitâche et parfois multiutilisateur ;

les programmes applicatifs dédiés à des taches particulières. Ils sont formés d'une série de commandes contenues dans un programme source écrit dans un langage « compris » par l'ordinateur.

Des langages de différents niveaux

Chaque processeur possède un langage propre, directement exécutable : le langage machine. Il est formé de 0 et de 1 et n'est pas portable, mais c'est le seul que l'ordinateur « comprend » ;

le langage d'assemblage est un codage alphanumérique du langage machine. Il est plus lisible que la langage machine, mais n'est toujours pas portable. On le traduit en langage machine par un assembleur ;

les langages de haut niveau, Souvent normalisés, permettent le portage d'une machine à l'autre. Ils sont traduits en langage machine par un compilateur ou un interpréteur.

Deux techniques de production des programmes

La compilation est la traduction du source en langage objet. Elle comprend au moins quatre phases (trois phases d'analyse -lexicale, syntaxique et sémantique et une de production de code objet). Pour générer le langage machine il faut encore une phase particulière : l'édition de liens. Cette technique est contraignante mais offre une grande vitesse d'exécution ;

Dans la technique de l'interprétation, chaque ligne du source analysé est traduite au fur et à mesure en instructions directement exécutées. Aucun programme objet n'est généré. C'est très souple mais l'interpréteur doit être utilisé à chaque exécution. . .

Ces deux techniques peuvent coexister pour un même langage.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL. . .
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL. . .
- Années 70 (génie logiciel) : C, MODULA-2, ADA. . .
- Années 80–90 (programmation objet) : C++, Labview, Eiffel, Matlab. . .
- Années 90–2000 (langages interprètes objet) : Java, Perl, tcl/Tk, Ruby, Python. . .

Parmi des centaines de langages crées, une minorité est vraiment utilisée.

Pourquoi un cours d' "Algo" ?

Objectifs: c'est de demander de la «machine» qu'elle effectue un travail à notre place, pour gagner plus de temps et de minimiser le risque d'erreur.

Problèmes:

- Comment expliquer et représenter cette demande à la «machine» sachant que cette dernière n'est qu'une machine.
- Mais comment représenter cette demande et la lui transmettre ?
- Comment elle va la traiter ?
- Comment s'assurer qu'elle fait ce travail aussi bien que nous ?

La résolution d'un problème informatique passe par quatre phases :

1- L'analyse
2- La conception
3- L'implémentation (La conception détaillée)
4- La validation

L’étape d’analyse

Cette étape permet de déterminer:

- Les objets(les données disponibles) liés à l’entrés (données du problème)
- Les objets liés à la sortis (résultats recherches)
- Les relations (éventuelles) entre les objets d’entrés et les objets de sortis qui peut être relations mathématiques, implications logique, …

La conception

Cette étape utilise le résultat obtenu de l'étape précédente(Analyse) pour le réécrire à l'aide d'un langage formel qui doit être:

- indépendant des langages informatiques,
- Ni de la machine qui exécutera le programme correspondant.

Le résultat de cette étape est appelé un **algorithme**

un **algorithme** est une suite d'actions que devra effectuer un automate pour arriver à partir d'un état initial, en un temps fini, à un résultat.

L’implémentation

La troisième étape:

Un programme est la, traduction d’un **algorithme** en un langage compréhensible par l’ordinateur (ou langage de programmation, ici le C). Le programme est ensuite exécuté pour effectuer le **traitement** souhaité.

La validation(test)

Cette étape sert à valider le programme (logiciel) par des tests sur plusieurs exemples.

STRUCTURE GENERALE D'UN PROGRAMME EN C

- Un programme en C se présente en général sous la forme suivante :

```
<Directives de compilation>
<Déclaration de variables externes>
<Déclaration de prototypes de fonctions>
main ( ){
    corps du programme
    (commentaires, déclaration de variables et constantes, instructions)
}
<Définition de fonctions>
```

- Exemple d'un programme en c :

```
#include <stdio.h>
#include <math.h>
double d ;  //variable globale (externe)
/* cette fonction reçoit les coefficients de l'équation et affiche les résultats possible */
 void   resolution(float a, float b, float c) ;
int main( ){   // fonction principale
      float a,b,c ;
      printf(''donner des valeurs à a, b et c'') ;
       scanf(''%f%f%f'', &a, &b ,&c) ;
      resolusion(a,b,c) ;
}
  void   resolution(float a, float b, float c) {
       double x1, x2  ;//variable locale à la fonction
       d=b*b-4*a*c ;
       if  (d>0) {    // la fonction  :   double sqrt(double )
                  x1=(-b-sqrt(d))/(2*a) ;
                  x1=(-b+sqrt(d))/(2*a) ;
                  printf(''les deux solution sont  %lf  et %lf'', x1,x2) ;
       }
       else  if (d==0)          printf (''la solution est. %f'',-b/a) ;
       else      printf("pas de solution dans R") ;
}
```

En langage c, un programme :

peut être composé par une ou plusieurs fonctions une et une seul doit porter le nom « main » (principale).

Une fonction peut appeler une autre

Une fonction peut appeler elle-même (récursivité)

Chaque variable utilisée doit être déclarée

Directives de compilation (include et define):

- Le langage c dispose d'un nombre important de bibliothèque de fonctions, l'appel à ces bibliothèques se fait par la ligne :

  #include < nom_fichier >

Exemple:

#include < stdio.h> /*ce fichier contient les fonctions d'entrées/sorties fichier comme printf*/

#include < conio.h> /*ce fichier renferme les routines d'entrée/sortie directes sur la console., kbhit(), clrscr() ….*/

#include < Math.h> //Ce fichier contient des fonctions mathématiques pouvant être appliquées aux types numériques.

#include <stdlib.h> : Gestion de la mémoire, conversions et fonctions systèmes

Le compilateur lit le contenu des fichiers .h et vérifie la syntaxe des fonctions appelées

- La directive define

  #define expression_de_remplacement expression_à_remplacer

permet de remplacer un symbole par une constante ou un type ou de faire des substitutions avec arguments dans le cas des macros.

Exemple:

```
#include <stdio.h> /*ce fichier contient les fonctions d'entrées/sorties comme printf*/
#define  pi 3.14   /*pi remplace la valeur 3.14*/
#define entier int /*entier remplace le type prédéfini int*/
#define    somme(x,y)  x+y /*la macro somme(x,y) remplace x+y*/
```

2. Pointeurs en langage C

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.

1- Introduction:

La Mémoire est un grand tableau de mots mémoire, chacun mots mémoires a un numéro qui le distingue des autres: ce numéro est appelé adresse.

C'est par cette adresse que le processeur peut communiquer avec la mémoire.

La taille du mot mémoire est un multiple de 8 bits.

On rencontre des ordinateurs ayant des mots mémoire de : 4 bits (historique: Intel 4004)8, 16 bits (autrefois)32, ou 64 bits (aujourd'hui)

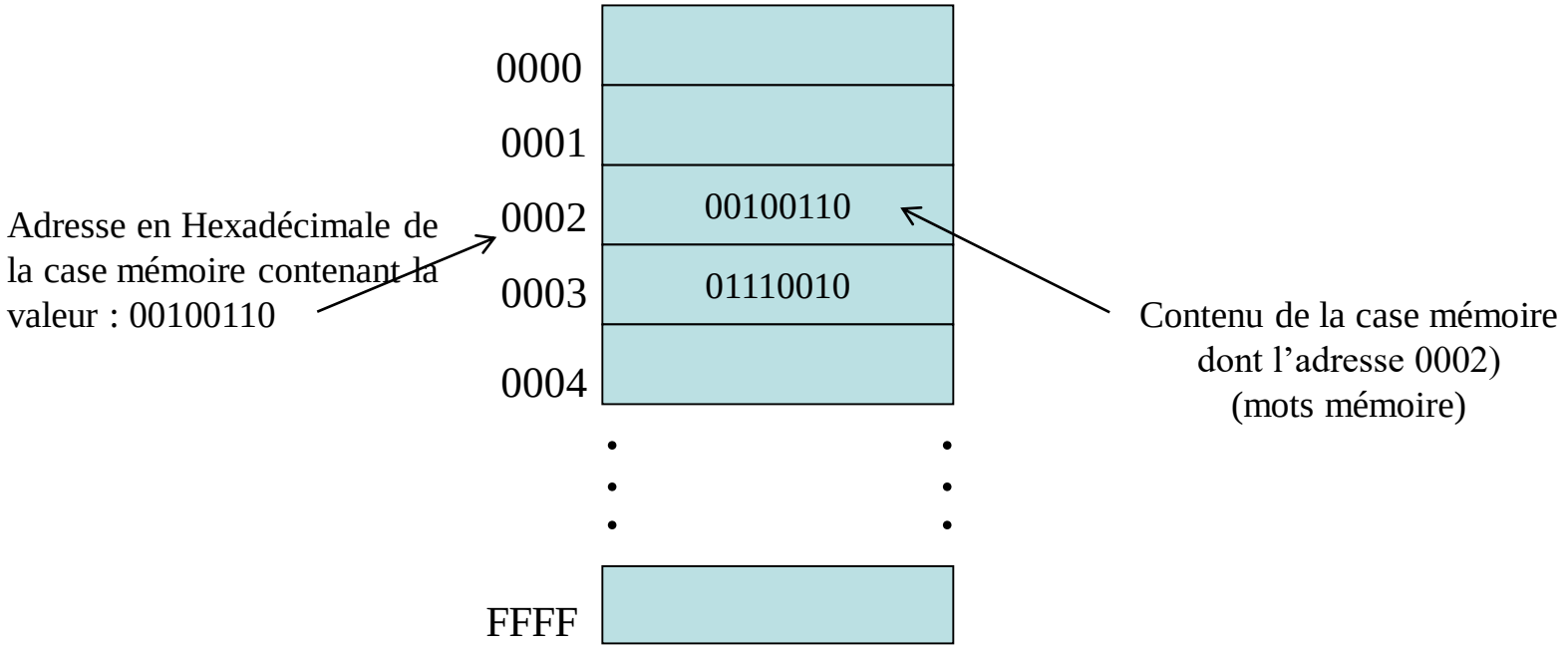


Figure 1: Structure d'une mémoire avec un mots mémoire de 8bits

Remarque:

Le nom qu'on donne à une variable lors de la déclaration se traduit en une adresse juste avant l'exécution du programme. Cela est nécessaire afin que le processeur sache à quelle case mémoire est associée chaque variable.

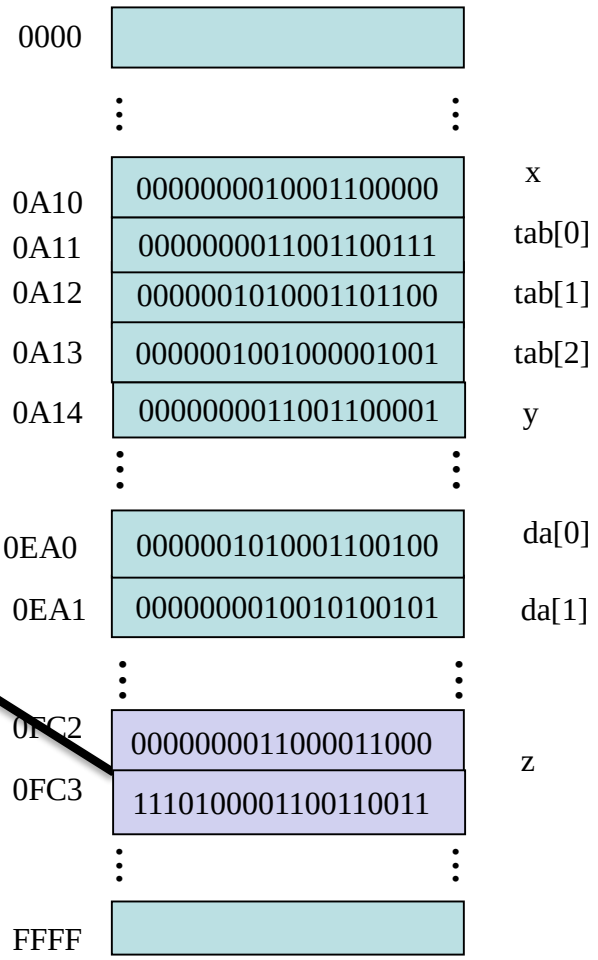
Exemple: on suppose que le mots mémoire est de 16bits et les entiers codés sur 16bits.

int x=1120, tab[3], y=1633, da[2];

float z; // 32 bits

Z= 818365235;

z 00000000110000110001110100001100110011



Remarque:

il n'y a aucune garantie que les variables occupent des cases adjacentes en mémoire Sauf pour les tableaux.

2- Définition d'un pointeur

Un pointeur est une variable ou une constante dont la valeur est une adresse. . Il s'agit d'une adresse en mémoire où sont stockées les données référencées.

L'adresse d'une variable est indissociable de son type. On pourra définir par exemple des pointeurs de caractères, des pointeurs d'entiers voir des pointeurs d'objets plus complexes (comme pointeurs sur des tableaux, des structures, …).

3-Intérêts des Pointeurs :

- Ils permettent de définir des structures dynamiques, c'est-à-dire qui évoluent au cours de l'exécution (par opposition aux structures de données statiques dont la taille est figée à la définition)
- Ils permettent l'accès par adresse (rapide) à la mémoire allouée.
- 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.

4- Déclaration d'une variable pointeur

Une variable de type pointeur se déclare à l'aide de l'objet pointé précédé du symbole *** (opérateur d'indirection).**

Syntaxe :

<Nom_type> *<ptr> ;

Nom_type: désigne le type de la variable

ptr : nom du pointeur.

(*) c'est un opérateur qui indique que le pointeur ptr pointe sur la variable de type (Nom_type)

Exemple:

char *pc; //pc est un pointeur pointant sur un objet de type char

int *pi, **ppi;//pi est un pointeur pointant sur un objet de type int et ppi pointeur sur un pointeur d'un entier

float *pr; // pr est un pointeur pointant sur un objet de type float .

5- Les opérateurs & et *

L'opérateur & est utilisé pour avoir l'accès à l'adresse d'une variable.

Syntaxe: &var /*retourne l'adresse de la variable var*/.

Exemple

```
int i=2 ;
printf(" voici  i :d\n ",i) ;
printf(" voici son adresse en hexadécimale : %x",&i) ;
```

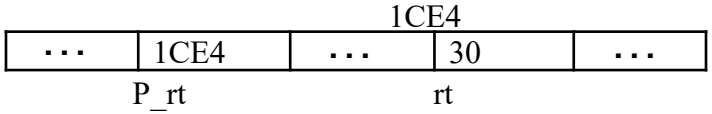
Après avoir déclaré un pointeur il faut l'initialiser. Pour initialiser un pointeur on utilise la syntaxe suivante :

Nom_du_pointeur=&nom_de_la_variable_pointee ;

Une fois l'adresse d'une variable est affecté à un pointeur, ce dernier peut être utilisé pour accéder aux cases mémoires correspondant à la variable à l'aide de l'opérateur * (valeur de la variable) :

Exemple 2

```
int rt=100, p ;
int *P_rt ;
P_rt =&rt ;
p=*P_rt ;  /* affecte à p la valeur désignée par * P_rt c-à-d ici la valeur de rt  (100)*/
* P_rt =30 ; /*affecte à l'entier pointé par P_rt la valeur 30 ceci équivalent à rt=30 */.
```



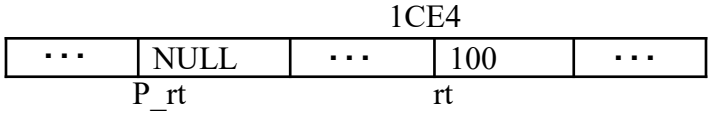
- Pointeur NULL

Pour indiquer qu'un pointeur pointe nulle part, on utilise l'identificateur NULL (On doit inclure stdio.h ou iostream.h).

Exemple 1

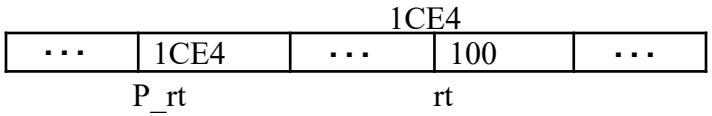
int rt=100 ;

int *P_rt=NULL ;



P_rt = & rt ; //on dit que P_rt pointe sur rt

D'où le schéma suivant :



6- Opérations sur les pointeurs

1- **Comparaison**

Condition : les pointeurs doivent être de même type.

EXEMPLE :

```
/* Déclaration de deux pointeurs sur des entiers */
        int* iP1=NULL, x, *iP2=&x;
/* Comparaison des adresses de iPointeur1 et iPointeur2, */
/* mais pas de leur contenu */
        if(iP1==iP2)    // vrai si les deux pointeurs pointent sur la même adresse et faux sinon
```

2- **Affectation**

```
iP1=iP2;  // Affecter l'adresse de iP2 au pointeur iP1
*iP1=*iP2;  // Affecter la valeur de la variable dont l'adresse dans iP2 à la variable
            dont l'adresse est dans le pointeur iP1
*ip1=NULL;
```

Remarque:

Le seul entier qui puisse être affecté à un pointeur d'un type quelconque P est la constante entière 0 désignée par le symbole NULL défini dans <stdio.h>.

On dit alors que le **pointeur P ne pointe nulle part.**

3- incrémentation et décrémentation :

On peut ajouter (ou soustraire) un nombre entier à la valeur d'un pointeur pour pointer sur un emplacement mémoire différent.

Exemple:

iP1++; // incrémentation de l'adresse stockée dans le pointeur iP1, si iP1=0200 alors après //l'instruction iP1++; iP1= 0200+sizeof(int);

//avec 0200 est l'adresse de début la zone mémoire réservée.

*iP1++; // incrémente la valeur dont l'adresse est stockée dans le pointeur iP1.

iP1--; // décrémentation de l'adresse stockée dans le pointeur iP1, si iP1=0200 alors après //l'instruction iP1--; iP1= 0200-sizeof(int);

iP1+n; //iP1=0200+n*sizeof(int), iP1 va pointer vers le nième élément.

Remarques

Eviter d'effectuer des opérations mathématiques comme des divisions, des multiplications ou des modulos sur les pointeurs.

Exercices

Indiquer les erreurs qui se trouvent dans les instructions suivantes :

a) **int *p , x = 34; *p = x;**

***p = x est incorrect parce que le pointeur p n'est pas initialisé**

b) **int x = 17 , *p = x; *p = 17;**

***p = x est incorrect. Pour que p pointe sur x *p = &x**

c) **double *q; int x = 17 , *p = &x; q = p;**

q = p incorrect. q et p deux pointeurs sur des types différent

d) **int x, *p=NULL; &x = p;**

&x = p incorrect. &x n'est pas une variable et par conséquent ne peut pas figurer à gauche de l'affectation.

e) **char mot[10], car = 'A', *pc = &car ; mot = pc;**

mot = pc incorrect. mot est un pointeur constant et on ne peut pas changer sa valeur. Ce n'est pas une variable .

3. Les pointeurs et tableaux (Allocation dynamique).

1- Application aux tableaux

Jusqu'à maintenant, lors de la déclaration d'un tableau, il fallait préciser les dimensions de façon explicite :

```
int mat[3][2] ,  tab[5] ;
```

soit de façon implicite :

```
int mat[][] = { {0 , 1 } , {2 , 3 } , {4 , 5 } }, tab[]={0,1;4,2,-1} ;
```

Dans les 2 cas, on a déclaré le tableau mat de 3 fois 2 entiers et le tableau tab de 5 éléments.

Mais si l'on veut que le tableau change de taille d'une exécution à une autre, cela nous oblige à modifier le programme et à le recompiler à chaque fois, ou bien de déclarer un tableau de 1000 fois 1000 entiers et n'utiliser que les *n* premières cases mais ce serait du gâchis.

Pour éviter cela, on fait appel à l'**allocation dynamique de mémoire** : au lieu de réserver de la place lors **de la compilation**, on la réserve pendant **l'exécution** du programme, de façon interactive. L'allocation dynamique se fait par des fonctions prédéfinies en langage c(malloc , calloc, realloc, free) qui sont définies dans la bibliothèque <stdlib.h> ,

- La fonction malloc()

Pour l'utiliser il faut inclure la bibliothèque <stdlib.h> en début du programme (#include <stdlib.h>).

malloc(N) renvoie l'adresse d'un bloc de mémoire de **N** octets libres adjacents (ou la valeur NULL s'il n'y a pas assez de mémoire à allouer).

(type *)malloc (nb-objets*nb-octets);

Exemple

```
int *p ;
p = (int *)malloc(800) ; /* fournit l'adresse d'un bloc de 800 octets libres  et l'affecte au
                                    pointeur p */
```

Le type **(int *)** devant **malloc** s'appelle un **cast**. Un cast est un opérateur qui convertit ce qui suit selon le type précisé entre parenthèses ici un **int**.

! Attention, il peut toujours se produire des erreurs lors de l'allocation dynamique de mémoire. il faut TOUJOURS vérifier que le pointeur retourné lors de l'allocation n'est pas NULL! **(impossible de trouver l'espace mémoire demandé)**

```
int *tab;
tab = (int *) malloc( 1000*sizeof(int) );
if( tab == NULL ){
printf("allocation ratée !");
}
else{
…
}
```

Principe général d'allocation:

1) Demande d 'allocation de l'espace mémoire
2) Vérification si la mémoire a été alloueé avec succès
3) Utilisation de la mémoire allouée

Remarque: pointeur sur un tableau statique

La déclaration suivante:

int tableau[100], *p_tableau ;

p_tableau=tableau ; ou p_tableau=&tableau[0] ;

Signifie que les éléments du tableau statique tableau sont parcourus par le pointeur p_tableau et non pas une réservation dynamique. Il faut alors utiliser l'une des fonctions d'allocation dynamique comme malloc() pour parler de l'allocation dynamique.

Incrémentation et décrémentation d'un pointeur:

*(p_tableau+1) désigne le contenu de tableau[1]

*(p_tableau+i) désigne le contenu de tableau[i] avec i<100

p_tableau =&tableau[i];

P_tableau+=n; désigne l'adresse de l'élément tableau[i+n]. (&tableau[i+n-1])

--P_tableau; désigne l'adresse de l'élément tableau[i+n-1]. (& tableau[i+n-1])

*(P_tableau+n) désigne le contenu de l'élément tableau[i+n]

L'opérateur sizeof()

D'une machine à une autre, la taille réservée pour un int, un float,... change. Si nous voulons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine á l'autre, nous avons besoin de la taille effective d'une donnée de ce type.

L'opérateur ***sizeof***() nous fournit la taille du type passé en argument.

int tabS[100]; // déclaration statique de 100 éléments de type entier.

int *tabD =(int *) malloc(100*sizeof(int)) ; // déclaration dynamique de 100 éléments.

tabD: l'adresse de début de la mémoire réservé(l'adresse du premier élément) et *tabD sa valeur.

tabD+1 : l'adresse du deuxième élément et *(tabD+1) sa valeur.

tabD+99: l'adresse du dernier élément et *(tabD+99) sa valeur.

int matS[3][4]; // déclaration statique d'un tableau à deux dimension de 3*4 éléments de type entier.

int *matD=(int *)malloc (N*M*sizeof(int)); // déclaration dynamique du même tableau de N*M éléments de type entier.

matD: l'adresse de début de la mémoire réservé(l'adresse du premier élément) et *matD sa valeur.

matD+N*i+j : l'adresse de l'élément d'indice i et j (0<=i<N et 0<=j<M)et *(matD+N*i+j) sa valeur.

- Fonction calloc:

Le même rôle que malloc mais elle permet de réserver **nb objets** de **nb octets** et **l'initialiser à 0**.

Syntaxe:

calloc(nb-objets, nb-octets);

```
int  *P=(int*)calloc(20, sizeof(int)); // allocation d'un tableau de 20 éléments de type entier
                                       //initialisées à zéro.
```

avec malloc:

```
int *P= (int*)malloc(20*sizeof(int));
for(int i=0; i<20; i++)    *(P+i)=0;
```

L'emploi de calloc est plus pratique et plus rapide.

- La fonction realloc()

realloc() sert à ré-attribuer de la mémoire à un pointeur mais pour une taille mémoire différente (augmenter ou réduire la mémoire).

realloc() s'utilise après l'allocation de la mémoire avec l'une des fonctions malloc() ou calloc(), et on peut la rappeler plusieurs fois dans le programme.

```
realloc(adresse_du_pointeur,  nouvelle-taille);
void* realloc (void *P,  int size_t  newsize);
int  *Ptmp;
Ptmp= (int *) realloc(P,  100*sizeof(int));
```

Noté Bien:

```
int * tab;
tab = (int *) calloc ( 2, sizeof(int) );
tab[0] = 33; tab[1] = 55;

tab = (int *)realloc(tab, 3 * sizeof(int) );
tab[2] = 77;
```

Quel risque peut on avoir si on opte pour l'appel précédent de realloc ????

Si la ré-allocation échoue, realloc renvoie NULL, qui est affecté à tab, on perd donc l'adresse de la zone mémoire allouée !

La solution consiste à créer un autre pointeur: temp par exemple, et c'est à temp qu'on affectera le résultat de realloc. Il suffit ensuite de tester si la valeur retournée par realloc n'est pas nulle on la réaffecte à tab sinon un message s'affiche pour indiquer que la ré-allocation a échoué.

```
temp = (int *)realloc(tab, 3 * sizeof(int) );
 if (temp != NULL)       tab = temp;
  tab[2] = 77;
```

Remarques:

- Si la zone mémoire précédemment allouée peut être augmentée sans empiéter sur une zone mémoire déjà utilisée par autre chose, alors realloc renvoie la même adresse. (l'adresse de la zone mémoire précédemment allouée)
- En revanche, dans le cas ou il y'a un débordement sur une zone mémoire déjà occupée, le système d'exploitation cherchera une autre adresse d'une zone mémoire de taille demandée. La fonction realloc() renvoie l'adresse du nouveau espace mémoire alloué.
- Si la ré-allocation échoue, realloc renvoie NULL,

- La fonction free()

La mémoire n'étant pas infinie, lorsqu'un emplacement mémoire n'est plus utilisé, il est important de libérer cet espace.

la fonction: **void free(void *p);**

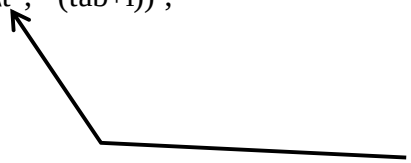
- libère l'espace mémoire pointé par p qui a été obtenu lors d'un appel à malloc, calloc ou realloc.
- Le pointeur p n'est pas remis à NULL, c'est au programmeur de le faire.

Exemple1

```
int *tab , n ;
printf("taille du tableau : n= ") ;
scanf("%d", &n) ;
tab = (int *)malloc(n*sizeof(int)) ;
……..
free(tab);
tab=NULL;
```

- **Cas d'un tableau à une dimension**

```
int n , *tab ;
printf("taille du tableau  n= ") ;
scanf("%d", &n) ;
tab = (int *)malloc(n*sizeof(int)) ;}
printf("Donner les éléments du tableau :") ;
for(i=0;i<n; i++)
        scanf("%d", tab+i) ;
printf("affichage des éléments du tableau :") ;
for(i=0;i<n; i++)     printf("%d\t", *(tab+i)) ;
free(tabD);
```



\t: constante de caractère équivalente à tabulation du clavier

Constantes:
a: alert
b: backspace
f: form feed
n: new line
r: carriage return
v: vertical tab

Exercice d'application:

Créer un tableau d'entiers dynamique T de taille N dans la méthode main.

- Lire les N éléments du tableau au clavier, d'indices allant de 0 à N-1
- Afficher le contenu de T
- Ajouter deux nouveaux éléments à la fin du tableau T sans avoir à écraser les données qui sont dans T.
- Afficher à nouveau le tableau

Pointeur et tableau dynamique à deux dimensions

Un tableau à deux dimensions (Matrice) est, par définition, un tableau de tableaux.

Méthode 1: *un pointeur vers un pointeur*

```
int  **M, L, C, i , j;
printf("entrer L=");  scanf("%d",&L);
printf("entrer C=");  scanf("%d",&C);
M = (int**)malloc(L * sizeof(int*));
for (i = 0; i < L; i++)
   M[i] = (int*)malloc(C * sizeof(int));
//Lecture au clavier
for (i = 0; i < L; i++)
      for (j = 0; j < C; j++)
            scanf("%d",M[i]+j);
// affichage du tableau
for (i = 0; i < L; i++){
      printf("\n");
      for (j = 0; j < C; j++)
            printf("%d\t",*(M[i]+j));
      }
```

Méthode 2 : *un pointeur simple*

```
int  *Mp,  L=2, C=3, i ,j;
printf("entrer L=");  scanf("%d",&L);
printf("entrer C=");  scanf("%d",&C);

Mp= (int*)malloc(L *C* sizeof(int*));

//Lecture au clavier
for (i = 0; i < L; i++)
    for (j = 0; j < C; j++)
        scanf("%d",Mp+L*i+j);
// affichage du tableau
for (i = 0; i < L; i++){
   printf("\n");
   for (j = 0; j < C; j++)
        printf("%d\t",*(Mp+L*i+j));
   }
```

- la fonction free() pour le cas d'un tableau à deux dimensions (*un pointeur vers un autre pointeur*)

pour libérer (dés-allouer) l'espace mémoire déjà alloué par la fonction malloc en utilise la fonction free() dans le cas de la méthode 1.

Exemple :

```
for (int i = 0; i < L; i++) {
    free(M[i]);
    M[i]=NULL
}
free(M);
M=NULL
```

- Pour le cas d'un pointeur simple(la méthode 2) déjà vu:

```
free(Mp);
Mp=NULL
```

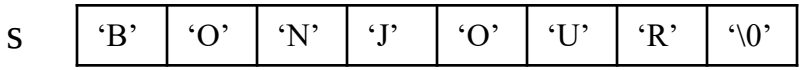
2- Application aux chaînes de caractères :

Une chaîne de caractères est un tableau de caractères. La déclaration de ce type de tableau entraîne une réservation de mémoire tout le long de l'exécution alors que la déclaration dynamique permet au programmeur d'élargir, de rétrécir ou de libérer un espace mémoire.

Syntaxe de déclaration statique

Char ms[nbrc] ; // ms est un tableau de nbrc caractères

Char ms[]=" bonjour ";



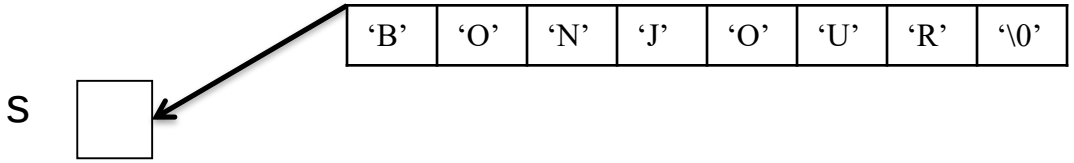
Syntaxe de déclaration dynamique

char *s ;

s=(char *)malloc(nbrc*sizeof(char)) ;// sizeof(char) retourne la taille du type char

s est un pointeur qui pointe sur un tableau de nbrc caractères. Le caractère '\0' marque la fin de la chaîne.

S= ''bonjour'' ;



Exemple(tester si une chaîne est un palindrome ou non)

```
int main(int arv, char  **ars){
      char *s ;
      int i,tr=0,n ;
      s=(char*)malloc(100) ;
      gets(s) ;
      n= strlen(s);
      for(i=0; i<n/2;i++)
            if(*(s+i)!=*(s+n-i-1){
                tr=1; break;}
            if(tr==0)
                printf("la chaine est un palindrome");
            else
                printf("la chaine n'est pas un palindrome");
       return 0;
 }
```

Remplacer la fonction gets par scanf et éxécuter le programme.

La différence entre **scanf** est **gets**:
- scanf permet de lire juste un mot s'arrête si elle tombe au cours de sa lecture sur un espace, une tabulation ou une entrée.
- par contre gets peut lire tout un tableau de chaine de caractères mais très dangereuse car elle ne permet pas de contrôler le buffer overflow ! (**dépassement de mémoire**)

- **Cas d'un tableau de chaines de caractères**

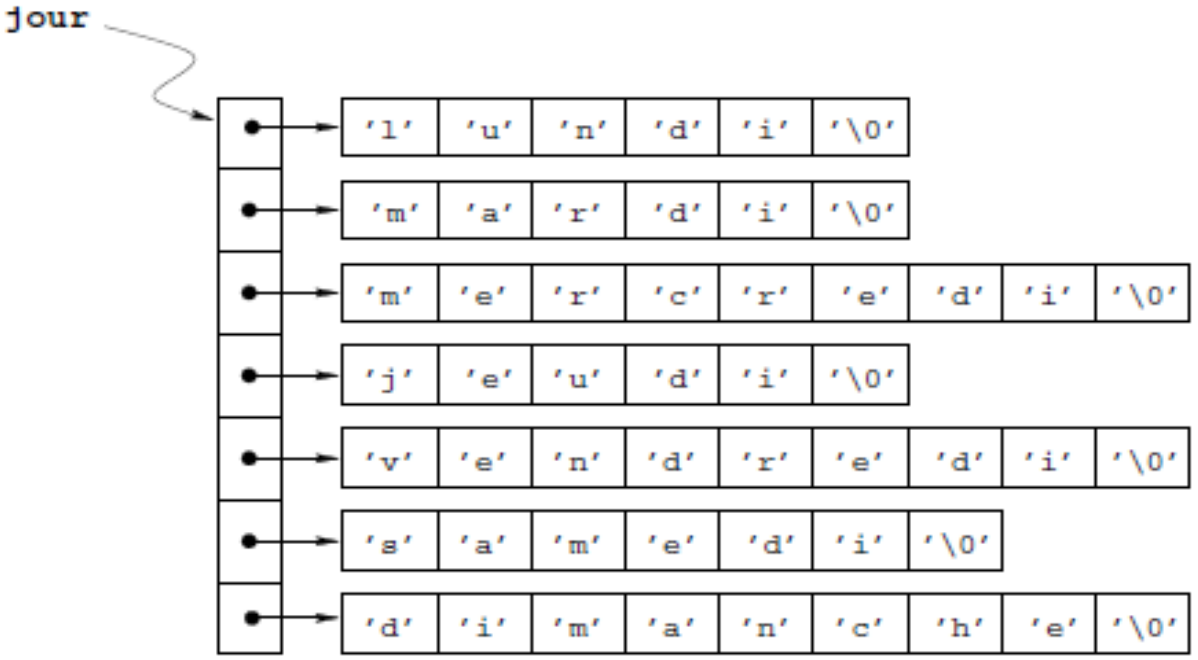
On peut initialiser un tableau de ce type par :

Char *liste[5]; //tableau de 5 éléments de type chaine de caractères

char * jour[] = {"lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"};

for (in i=0 ; i < NB_JOUR; i++) printf("%s\n",jour[i]);

L'allure du tableau jour est illustre dans la figure suivante:



Exemple:

```
char **ms;
ms=(char **)malloc(4*sizeof(char*)); // tableau de chaines de caractères de 4 éléments
for(int i=0; i<4;i++)        *(ms+i)=(char *)malloc (20*sizeof(char));   //chaque élément est de 20 caractères
for(int i=0; i<4;i++)        gets(ms[i]);
for(int i=0; i<4;i++)
      printf(" avec gets %s\n",*(ms+i));
```

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;
  . . .
```

Version 1:

```
int i=0
while((ch1[i]=ch2[i])!='\0') i++;
```

Version 2:

```
int i=0
while((*(ch1+i)=*(ch2+i))!='\0')   i++;
```

Le même code avec la notion pointeur

```
while((*ch1=*ch2)!='\0'){
        ch1++;
        ch2++;
}
```

- Traitements sur les chaînes de caractères

Les fonctions de traitements de chaînes sont définies dans l'en-tête <string.h>. Les principales sont :

- fonction **strlen** retourne la langueur d'une chaine passée en argument : strlen(str)
- fonction **strcmp** permet de comparer l'égalité de deux chaines strcmp(s1, s2) : retourne 0 si s1=s2, valeur >0 si s1>s2 et valeur<0 si s1<s2;

- fonction **strcpy** recopie une chaine dans une autre ;
- fonction **strcat** permet la concaténation de deux chaines
- fonction**, *strdup (const char * s):** Renvoie un pointeur sur une nouvelle chaîne de caractères qui est dupliquée depuis s, ou NULL s'il n'y avait pas assez de mémoire.

……

Il existe **des macros** expressions définies dans **< ctype.h >** qui permettent de déterminer ou de changer le type d'un caractère. Ces macros expressions de test retournent un résultat non nul si le test est vrai.

- isalpha(c) vrai si c est une lettre (alphabétique).
- isupper(c) vrai si c est une majuscule (upper case).
- islower(c) vrai si c est une minuscule (lower case).
- isdigit(c) vrai si c est un chiffre.

- isspace(c) vrai si c est un blanc, interligne ou tab.
- ispunct(c) vrai si c est un caractère de ponctuation.
- isalnum(c) vrai si c est alphabétique ou numérique.
- isprint(c) vrai si c est affichable de 040 à 0176.
- isgraph(c) vrai si c est graphique de 041 à 0176.
- iscntrl(c) vrai si c est del (0177) ou un caractère de contrôle (<040).

Les macros qui transforment un caractère :

– toupper(c) retourne le caractère majuscule correspondant à c.

– tolower(c) retourne le caractère minuscule correspondant à c.

Les fonctions de lecture et affichage:

♦ La fonction **gets** retourne **l'adresse de la chaîne lue**. Tous les caractères sont acceptés (même les espaces). Le \0 est mis à la fin de la chaîne, mais il n'y a toujours pas de contrôle de longueur.

♦ La fonction **puts** affiche **la chaîne** qui est en argument, jusqu'à ce que la fonction rencontre un \0, et passe à la ligne.

♦ Les fonctions int getch() et int getchar () **permet la lecture d'un seul caractère du clavier**. La fonction **getch()** lit le caractère **sans affichage** sur l'écran et **l'autre** fonction lit le caractère mais avec affichage.

Ecrire les fonctions qui:

1) Permet d'allouer un espace pour un tableau dynamique de N chaines de caractères dynamiques
2) Permet la saisie au clavier d'un tableau dynamique de N chaines dynamiques
3) Permet l'affichage d'un tableau dynamique de N chaines dynamiques
4) Permet de concaténer dans une nouvelle chaine dynamique (à retourner) les N chaines du tableau.
5) Refaire la question 4) sachant que la concaténation se fait dans la première chaine du tableau
6) Ecrire un programme principal de test.

4. Les fonctions et pointeurs.

Les fonctions en C

1. La notion de fonction

On appelle **fonction ou** sous programme, **une partie de code source** qui permet d'effectuer un ensemble d'instructions.

Une fonction désigne une entité de données et d'instructions qui fournissent une solution à une (petite) partie bien définie d'un problème plus complexe. **Une fonction peut faire appel à d'autres modules,** leur transmettre des données et recevoir des données en retour. L'ensemble des fonctions ainsi reliés doit alors être capable de résoudre le problème global.

La plupart des langages de programmation nous **permettent de subdiviser** les programmes en sous-programmes, (**fonctions ou procédures)** plus simples et plus compacts.

Avantages :

Voici quelques avantages d'un programme fonctionnelle (modulaire):

*** Meilleure lisibilité**

*** Diminution du risque d'erreurs**

* **Possibilité de tests sélectifs**

* **Dissimulation des fonctions:** Lors de l'utilisation d'une fonction il faut seulement connaître son effet, sans devoir s'occuper des détails de sa réalisation.

* **Réutilisation de fonctions déjà existantes:** Il est facile d'utiliser des fonctions qu'on a écrits soi-même ou qui ont été développés par d'autres programmeurs.

* **Simplicité de l'entretien:** Un module peut être changé ou remplacé sans devoir toucher aux autres modules du programme.

* **Favorisation du travail en équipe** Un programme peut être développé en équipe par délégation de la programmation des fonctions à différentes personnes ou groupes de personnes. Une fois développés, les fonctions peuvent constituer une base de travail commune.

* **Hiérarchisation des fonctions**

2. La déclaration de signature (prototype) d'une fonction

Avant d'être utilisée, une fonction doit être déclarée. La déclaration d'une fonction (ou signature) se fait selon la syntaxe suivante :

[Modificateur] <TypeRetour> <Nom_Fonction> (<type1> <arg1>, <type2> <arg2>,……..)

TypeRetour : représente le type de la valeur que la fonction est sensée retourner (char, int, float…)

Nom_Fonction : représente le nom donné à la fonction

type1, type2, …….sont les types respectivement des arguments arg1, arg2, ……..

Remarques :

- **L'imbrication d'une fonction n'est pas autorisée en C,** une fonction ne peut pas être déclarée à l'intérieure d'une autre fonction. Par contre une fonction peut appeler une autre.
- Si **la fonction ne renvoie aucune valeur, on** la fait précéder du mot-clé void (procédure dans d'autre langage)

  syntaxe **: void nom_fonction(arguments)**
- Fonctions renvoyant une valeur au programme **et sans arguments**

  Syntaxe **: Type nom_fonction( ) ;**
- Si **aucun type** de données n'est précisé, le **type int** est pris par défaut
- Les arguments sont facultatifs, mais s'il n'y a pas d'arguments **les parenthèses doivent figurer dans la définition.**

Exemples de signature de fonctions

int plus(int a, int b) ;

La fonction **plus** retourne un résultat de type entier et qui a deux arguments a et b de type entiers aussi.

void add(int a, int b, int*c) fonction qui ne retourne pas de résultat et qui a trois arguments : a et b sont deux entiers et c un pointeur sur un entier.

3. La définition du corps d'une fonction

Une fonction comporte deux parties :

La signature de la fonction et le corps. Le corps contient des déclarations de variables locales et un ensemble d'instructions indiquant ce que la fonction doit faire. Il est sous forme d'un bloc.

Syntaxe

```
Type_f   nom_f  (type1  p1, ……,typen  pn)   //signature de la fonction
   {
          variables locales
          Instructions        //corps de la fonction
   }
```

Avec :

type_f : type de la valeur retournée par la fonction

nom_f : nom de la fonction

$type_i$ ($1<i<=n$) : type du i^{eme} argument de la fonction

p_i ($1<i<=n$) : nom du i^{eme} argument (paramètre) de la fonction.

Exemple :

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

Remarques :

- Le rôle de l'instruction return dans la fonction est **double** : d'une part, il précise la valeur qui sera fournie en résultat, d'autre part, **il met fin à l'exécution** des instructions de la fonction.

- la durée de vie des variables locales **est limitée** à celle de l'exécution de la fonction. Leur portée (ou espace de validité) est donc limitée à cette fonction.

- **On distingue trois possibilités de déclarer et de définir les fonctions** y compris la fonction main dans les programmes en C:

a – Définition 'bottom-up' sans déclaration

Déclaration des bibliothèques à utiliser et variable globales

```
Type retour f1(arguments){
……
}
Type retour f2(arguments){
……
}
...
int main( ){
    instructions de déclaration
    instructions exécutables
return 0 ;
}
```

b- Déclarations globales des fonctions 'top-down'

Déclaration des bibliothèques à utiliser et variable globales

```
Type_retour f1(arguments);
Type_retour f2(arguments);
.....
int main( ){
    Instructions de déclaration
    Instructions exécutables
return 0 ;
}
Type_retour f1(arguments){
……
}
Type_retour f2(arguments){
……
}
```

c- Déclarations locales des fonctions 'top-down'

Déclaration des bibliothèques à utiliser et variable globales

```
int main( ){
        Type_retour f1(arguments);
        Type_retour f2(arguments);
        instructions de déclaration
        instructions exécutables
        return 0 ;
}
Type_retour f1(arguments){
        ……
}
Type_retour f2(arguments){
        ……
}
...
```

4 – Appel d'une fonction

Pour exécuter une fonction, il suffit de l'appeler en écrivant son nom suivie de la liste des arguments.

Syntaxe

```
Nom_De_La_Fonction(arguments) ;
```

Exemple

```
#include<stdio.h>
int somme(int x, int y);
void main() {
      int  x, y ;
      printf("Entrer les valeur à sommer : ") ;
       scanf("%d %d ",&x,&y) ;
      printf("\n la somme de %d et %d est égal à %d ", x, y, somme(x, y)) ;
}
 int somme(int x, int y) {
          return(x+y) ;
      }
```

5. Passage des arguments à une fonction : (Utilisation des pointeurs)

Le passage d'arguments à une fonction peut se faire par valeur , par référence ou par adresse (pointeur)

- Dans le passage par valeur, juste une copie de la valeur est transmise à la fonction(les modifications apportées à l'intérieur de la fonction ne se répercutent pas sur les autres fonctions).
- Dans le passage par référence, une adresse de la valeur est transmise à la fonction(les modifications effectuées à l'intérieur de la fonction se répercutent en dehors de la fonction).

Exemple .cpp :

```
void echanger(int  &, int *) ;
 int n=2, m=3 ;
int  main(   )   {
      int *x; x=&m;
      printf("\n n=%d   et m=%d",n,m);
      echanger(n, x) ;
      printf("\n après  n=%d   et m=%d",n,*x);
return 0;
}
    void echanger(int  &a, int *b) {
                int k ;              /*valeur intermidiare*/
                k=a;
                a=*b;
                *b=k;
      }
```

- Appel de la fonction echanger() avec passage par référence de l'argument n et par adresse de l'argument m.
- l'opérateur & obligatoire pour m et non pour n.

Variables locales et globales :

- Une variable connue dans tout le programme est une variable **globale** (déclarée au début du programme en dehors de toute fonction y compris main).

  En C **une variable globale est partagée par plusieurs fonctions** (d'un même fichier), elle peut même être partagée entre plusieurs fichiers sources.

- Une **variable locale** à une fonction **n'est accessible que par cette fonction**(exemple précédent: la variable k locale à la méthode echanger()). Les valeurs des variables locales ne sont pas conservées d'un appel au suivant et ont une "durée de vie" limitée à celle d'une exécution de la fonction dans laquelle elles figurent.

- **Une variable locale statique** peut être déclarée à l'intérieur d'une fonction mais possède un comportement de visibilité particulier.

  On peut dire qu' Une variable locale **statique** c'est le **fait d'attribuer un emplacement permanent** à une variable locale

  Exemple: **static int c;**

Le mots **static** indique que la valeur de la variable **sera persistante entre les différents appels de la fonction** où elle est définie. La variable ne sera visible que dans la fonction, **mais ne sera pas réinitialisée à chaque appel de la fonction.**
L'intérêt est de **garantir une certaine encapsulation, afin d'éviter des usages multiples d'une variable globale** et permet d'avoir aussi plusieurs fois le même nom dans des fonctions différentes.

Variable static locale:

```
#include<stdio.h>
    int fun() ;
int main() {
  printf("%d ", fun());
  fun();
  printf(",%d ", fun());
  return 0;
}
 int fun() {
  static int count = 0;
  count++;
  return count;
}
```

Sortie : 1, 3

Variable locale non statique:

```
#include<stdio.h>
 int fun() ;
int main() {
  printf("%d ", fun());
  fun();
  printf(",%d ", fun());
  return 0;
}
int fun() {
  int count = 0;
  count++;
  return count;
}
```

Sortie :1, 1

Exercice:

1) Ecrire une fonction void saisie(int * T, int taille), qui reçoit en argument le tableau dynamique T et sa taille et qui permet d'allouer l'espace mémoire et lire au clavier ses éléments.
2) Ecrire une fonction Affiche(int * T, int taille), qui affiche les éléments du tableau T.
3) Ecrire un programme principal de test.

6. Pointeur de fonction

Une fonction peut retourner une valeur , une adresse ou un pointeur sur la fonction.

- Syntaxe d'une fonction qui retourne une valeur:

<type de Retour> <nomFonction>(arguments);

- Syntaxe d'une fonction qui retourne une adresse:

<type de Retour>* <nomFonction>(arguments);

- Syntaxe de la définition d'un pointeur sur une fonction:

type (*<identificateur>) (paramètres);

Exemple: void (*ptr_fonction) (int, char);

La variable « ptr_fonction » est un pointeur sur une fonction qui a 2 arguments du type (int, char) et ne retourne rien.

Exemple:

- Une fonction qui retourne une adresse de type « int ».

```
int *Fonction();
```

- La variable « ptrfonction» est un pointeur sur la fonction Fonction () qui n'a aucun argument et qui retourne un « int » (la fonction **Fonction()** est supposé déjà définie):

```
int (*ptrfonction )();
ptrfonction= fonction;
```

Remarques:

- En C, il est possible de référencer des instructions à l'aide des pointeurs de fonction.
- Ne pas confondre pointeur sur une fonction et une fonction qui retourne un pointeur.

Exemple:

```
void echange(int& , int*);
int main(int argc, char** argv) {
      int *b,i=2,j=200;
       b=&j;
      void (*pointeurSurFonction)(int& a, int *b); /*déclaration du pointeur*/
      printf("avant l'echange a= %d   b=%d\n",i,*b);
      pointeurSurFonction = echange;              /*Initialisation*/
     (*pointeurSurFonction)(i,b);                 /*Appel de la fonction*/
      printf("après l'echange a= %d   b=%d",i,*b);
      return 0;
}
void echange(int& a, int *b) {
      int k;                  /*valeur intermidiare*/
      k=a;
      a=*b;
      *b=k;
}
```

7. Les fonctions récursives

Définition :

Une fonction récursive est une fonction qui s'appelle elle-même directement (récursivité simple) ou indirectement (récursivité mutuelle)

Récursivité directe ou simple :

Dans ce cas, une fonction comporte dans sa définition au moins un appel à elle-même.

Exemple :

```
int f(int n){
    int k ;
    if (n==0)    return 1 ;
    else{     k=n*f(n-1);     return k;      }
```

Récursivité croisée (indirecte) : L'appel se fait, lorsque deux fonctions ou plusieurs s'appellent l'une l'autre récursivement.

Exemple:

```
int pair ( int n ){
    if ( n == 0 )   return ( 1 );
    else
       return  ( impair ( n – 1));
}
int impair ( int n ){
    if ( n == 0 ) return ( -1) ;
    else    return ( pair ( n - 1 ));
}
```

Il y a deux notions à retenir :

La fonction s'appelle elle-même : on recommence avec de nouvelles données.

Il y'a un test de fin, dans ce cas (lorsqu'il est vérifié), il n'ya pas d'appel récursif. Il est souvent préférable d'indiquer le test de fin des appels récursifs en début de fonction.

Remarque : Tout usage de récursivité nécessite un test d'arrêt, sinon on aura une récursivité sans fin (jusqu'à saturation de la mémoire). Comme par exemple :

```
int   f(int x) {
      x++ ;  return (f(x)) ;
}
```

Exemples de fonctions récursifs :

Exemple: Factoriel :

Le factoriel d'un nombre N donné, est le produit des nombres entiers inférieurs ou égaux à ce nombre n.

La fonction factoriel(n) permet de calculer le factoriel de n d'une maniérer récursive.

Il y'a fin des appels récursifs lorsque n=0.

```
long factoriel(int n){
    if (n= =0) return 1 ;
    else
      return n*factoriel(n-1);
 }
```

	1	*2*	*3*	*4*
factorielle(3)	*n=3* *return3*factorielle(2)*	*n=2* *return2*factorielle(1)*	*n=1* *return1*factorielle(0)*	*n=0* *return(1)*

n=0➔ test vérifié ➔ arrêt de récursivité → **on revient en arrière**.

Remarque :

La récursivité est un moyen très simple et très rapide pour effectuer des taches complexes. Mais consomme beaucoup d'espaces mémoires pour sauvegarder les traces des niveaux d'appels récursives.

8- Fonctions avec un nombre variable de paramètres

En C, il est possible de définir des fonctions qui ont un nombre variable de paramètres.

Une fonction qui possède **un nombre variable de paramètre** doit obligatoirement posséder **au moins** un paramètre formel fixe suivie de le la notation **...** . Les trois points (…) indique que la fonction possède un nombre quelconque de paramètres.

Exemples:

```
int printf(char *format, ...);
int fonc(int a, ...);
```

Un appel à une fonction ayant un nombre variable d'arguments s'effectue de la même manière qu'une fonction avec un nombre fixe de paramètres.

Pour **accéder à la liste** des paramètres de l'appel, **on utilise les macros** définies dans le fichier en-tête **stdarg.h** de la librairie standard qu'il faut inséré .

Dans la fonction ayant un nombre variable d'arguments, Il faut tout d'abord déclarer une variable pointant sur la liste des paramètres de l'appel avec le type **va_list**.

```
va_list  list_argu;
```

La variable list_argu doit être initialisée à l'aide de la macro **va_start**, selon la syntaxe suivante:

```
va_start(list_argu, dernier_argument);
```

où dernier_argument **désigne l'identificateur** du dernier paramètre formel fixe de la fonction

Après traitement des paramètres, on libère la liste à l'aide de la **va_end :**

va_end(liste_argu);

L'accès aux différents paramètres de la liste se fait par la macro **va_arg** qui retourne le paramètre suivant de la liste:

va_arg(liste_argu, type)

où type est le type de l'argument auquel on accède.

Implémentation d'une fonction avec un nombre d'arguments variable :

```
#include <stdarg.h>
 void fonc (int n, ...) {
        va_list   list_argu;
        va_start (list_argu, n);
              /* traitement des paramètres selon la fonction */
         va_end (list_argu);
}
```

Exemple:

```
#include <stdarg.h>

int somme(int nb,...);

int main(void) {
  printf("\n %d", somme(4,90,4,2));
  printf("\n %d", somme(7,15,25,13,7,6,5));
  return(0);
}
int somme(int nb,...) {
  int res = 0;
  int i;
  va_list list_argu;
  va_start(list_argu, nb);
  for(int i=0; i<nb;i++){
      res+=va_arg(list_argu, int);
  }
  va_end(list_argu);
  return(res);
}
```

Remarque:

Tout usage de la récursivité est un moyen très simple et très rapide pour effectuer des taches complexes. Mais consomme beaucoup d'espaces mémoires pour sauvegarder les traces des niveaux d'appels récursives.

5. Les types de variables complexes(enregistrements, unions, énumérés)

1. Définition:

Un tableau permet de regrouper des éléments de même type, c'est-à-dire codés sur le même nombre de bit et de la même façon. Toutefois, il est généralement utile de pouvoir rassembler des éléments de type différents.

Les structures permettent de remédier à cette limite des tableaux, en regroupant des objets (des variables) au sein d'une entité repérée par un seul nom de variable.

Les objets (variables) contenus dans la structure sont appelés champs de la structure.

Le type structure est souvent manipulé dans les structures de données suivantes :

- tableau de structures
- listes chaînées, , arbre binaire
- fichier non texte.
- classe (programmation orientée objet)

etc ...

2. Déclaration d'une structure en C

Le langage C permet de déclarer une structure de la façon suivante :

```
struct <nom_de_struct> {
        <Type1> <nom_champ1> ;
        <Type2> <nom_champ2> ;
          …..   …..   ……. ;
        <TypeN> <nom_champN> ;
} ;
```

```
struct <nom_de_struct> {
        <Type1> <nom_champ1> ;
        <Type2> <nom_champ2> ;
          …..   …..   ……. ;
        <TypeN> <nom_champN> ;
} <nomVar1>, *<nomVar2>,…. ;
```

Les types des champs type1, type2,…, typeN peuvent être différents, identiques ou bien composés.

Exemples :

```
struct Date      {
      int jour ;
      int mois ;
      int an ;
      } ;
 struct Etudiant {
        int codeCNE;
        char Nom[12];
        float Moyenne;
        struct Date dateNaissance;
      } ;
```

Déclaration séparée de structure et de variables

Il est possible de déclarer séparément la structure et les variables associées. Dans ce cas, il faut d'abord déclarer la structure puis les variables après.

Déclaration des variables de type structure en même temps de la définition de sa définition

Exemples :

```
struct  Date       {
      int jour ;
      int mois ;
      int an ;
      } arrivee, dateNaissance;
```

Syntaxe de déclaration de variable structure séparée de la définition :

struct nom_struct var1, var2,…….;

Exemple :

Une fois la structure date déclarée, on peut écrire les déclarations suivantes :

struct Date depart, arrivee, *naissance ;

3. Utilisation de structures

Les structures peuvent être manipulées champ par champ ou dans leur ensemble.

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. La désignation du champ se note en faisant suivre le nom de la variable structure d'un point(ou de la flèche -> si la variable est de type pointeur) puis du nom de champ.

La syntaxe : **nom_de _variable . nom_du_champ**

nom_de _variableP -> nom_du_champ

Exemple :

En utilisant la déclaration

```
struct date {
        int jour ;
        int mois ;
        int an ;
} depart, arrivee,  *naissance ;
```

Pour accéder aux champs des variables départ, on écrira :

```
depart.jour =16 ;      depart.mois=2 ;     depart.an=2019;
arrivee.jour=28;       arrivee.mois=5;     arrivee.an=2019;
naissance ->jour=2;    naissance ->mois=3;   naissance ->an=2019;
```

4. Types synonymes

La principale utilité des typedef est de faciliter l'écriture des programmes, et d'en augmenter la lisibilité.

typedef sert à créer des types synonymes, autrement dit, créer de nouveaux types de données à partir d'un type existant (simple ou composé).

Exemple:

typedef int entier; /* On définit un nouveau type « entier » synonyme de int*/

Typedef int vecteur[3]; ; /* On définit un nouveau type « vecteur » synonyme d'un tableau de 3 entiers */

Exemple d'application: Définir le type BOOLEAN qui n'existe pas en c

#define VRAI 1

#define FAUX 0

typedef int BOOLEAN;

On pourra par la suite déclarer des << BOOLEAN>> de la manière suivante :

BOOLEAN b1,b2; et les utiliser :

b1 = VRAI; if (b2 == FAUX) ...

mais bien entendu, ce sera à la charge du programmeur d'assurer que les variables b1 et b2 ne prennent comme valeurs que VRAI ou FAUX. Le compilateur ne protestera pas si on écrit :b1 = 10; On voit que la lisibilité du programme aura été augmentée,

typedef **struct date Date;**

typedef **struct empolye Emp;**

Remarque:

Il est possible d'affecter à une structure le contenu d'une autre structure défini à partir du même modèle.

Ainsi si les variables s1 et s2, de type structure, sont définies suivant le même modèle nous pouvons écrire s1=s2 ;

Exemple:

```
struct date {
        int jour ;
        int mois ;
        int an ;
    } depart, arrivee,
depart.jour =3;  depart.mois=1;    depart.an=2018;          arrivee= depart;
```

Exercice d'application:

Réaliser un modèle de structure pour représenter des points dans le plan. Chaque point est caractérisé par son nom (caractère), son abscisse et son ordonné (des réels).

Ecrire les fonctions suivantes :

- Une fonction qui permet de saisir un point (ses caractéristiques)
- Une fonction qui affiche les caractéristiques (son nom, et ses coordonnées) d'un points passé en argument à la fonction.

Ecrire le programme principal de test.

Déclaration d'un tableau statique de structure

Il est toujours possible de déclarer des tableaux de type structure.

```
struct   <nom_de_struct>    {
        <Type1>  <nom_champ1> ;
        <Type2>  <nom_champ2> ;
          …..     …..     ……. ;
        <TypeN>  <nom_champN> ;
  } varTab [10], v;
```

Ou bien :

```
struct   <nom_de_struct>   varTab[10], v;
```

Exemple:

```
 struct Etudiant  {
        int codeCNE;
        char Nom[12];
        float Moyenne;
} liste[10];
```

Liste[0].codeCNE=123;

Liste[0].Nom="Aziz TAZI";

Liste[0].Moyenne=12,3;

Liste[1].codeCNE=124;

Liste[1].Nom="Samira ALAMI";

Liste[1].Moyenne=14,3;

……

5. Allocation mémoire des pointeurs sur les structures

Il faut toujours faire attention à la gestion dynamique des structures, ainsi que leurs champs. Possible de trouver des champs dynamiques.

Exemple:

Définition de la structure

```
struct  Etudiant  {
        int codeCNE;
        char *Nom;    // Nom est un pointeur sur un tableau de caractères
        float Moyenne;
    } ;
```

Déclaration d'un pointeur simple sur la structure Etudiant

Struct Etudiant *listEtud;

listEtud = (struct Etudiant*)malloc(sizeof(struct Etudiant)); /* *Allocation mémoire du pointeur listEtud sur la structure Etudiant* */

(* **listEtud**).Nom = (char*)malloc(50); /* *Allocation mémoire du champ Nom de la structure* */

listEtud->Nom=« Aziz»;

ou bien

(*listEtud).Nom=« Aziz»;

Déclaration dynamique d'un pointeur sur la structure Etudiant

```
Struct  Etudiant *listEtud;
listEtud = (struct Etudiant*)malloc(4* sizeof(struct Etudiant));
    /* Allocation mémoire du pointeur listEtud sur la structure Etudiant composé de 4 éléments */
for(int i=0;i<4;i++)
    (listEtud +i)->Nom = (char*)malloc(50);
        /* Allocation mémoire du champ Nom de la structure */
```

Exemple:

```
struct point{
        int x;    int y;    char *nom;
};
void lecturest(struct point *t, int n);
void affichage(struct point *t, int n);
int main(int argc, char *argv[]){   // fonction principale
   int i;    struct point *tv;
   tv=(struct point *)malloc(4*sizeof( struct point));
   for(i=0;i<4;i++)
         (tv+i)->nom=(char *)malloc(20*sizeof(char));
   lecturest( tv, 4);
   affichage( tv, 4);
    for(i=0;i<4;i++)    free ((tv+i)->nom);
    free(tv);
    return 0;
}
void lecturest(struct point *t, int n){
        int i;
        for(i=0; i<n ; i++){
                    scanf("%d",&(t+i)->x);
                    scanf("%d",&(t+i)->y);
                    scanf("%s",(t+i)->nom);
        }}
```

```
void affichage(struct point *t, int n){
   int i;
   for(i=0; i<n; i++){
      printf("(%d,  ",(t+i)->x);
      printf("%d, ",  (t+i)->y);
      printf("%s)\n", (t+i)->nom);
}}
```

Les champs de bits

Il est possible en C de spécifier la longueur des champs d'une structure au bit prés si ce champ est de type entier (int ou unsigned int). Cela se fait en précisant le nombre de bits du champ avant le ";" qui suit sa déclaration :

type <nom_de_la_variable> : nb_bits ;

On utilise généralement les champs de bits en programmation système, pour manipuler des registres particuliers de la machine mais ils peuvent être utilisés aussi dans les structures conventionnelles.

Par exemple, d'un registre qui peut se décrire de la structure suivante :

```
struct registre {
    unsigned int masque : 2;
    signed int privilege : 6;
    unsigned int no: 6;
    unsigned int ov : 1;
};
```

Le champ masque sera codé sur 2 bits, privilege sur 6 bits etc... Le champ ov de la structure ne peut prendre que les valeurs 0 ou 1. le champ masque ne peut prendre que 0, 1 , 2 et 3.

l'opération rg.ov += 2 ; ne modifie pas la valeur du champ.(rg est un objet de type struct registre)

Les contraintes à respecter avec les champs de bits :

– La taille d'un champ de bits doit être inférieure au nombre de bits d'un entier (long).

– un champ de bits n'a pas d'adresse ; on ne peut donc pas lui appliquer l'opérateur &.

6. Les unions

L'union est la déclaration de différentes variables qui occupent toutes la même place en mémoire. Le système alloue un emplacement mémoire tel qu'il pourra contenir la variable de plus grande taille

La syntaxe de déclaration d'une union est la suivante :

```
union     nom_union  {
      Type1   var1 ;
       Type2   var2 ;
       Type3  var3 ;
       …..     ……. ;
      TypeN   varN ;
}   u ;
```

Cette déclaration réserve un emplacement qui pourra contenir l'une des variables var1, var2, …. varN.

Pour accéder à l'une des variables de l'union, on utilise la syntaxe :

nom_union.nom_variable ;

Exemple :

```
union u  {
      char c ;
      float f ;
      long l ;}  j;
   j.c='a';
   j.f=15.4;          /* après cette déclaration, on a plus le caractère 'a' dans c */
```

On peut dire que l'union nous permet de représenter des variables qui change de type.

7. Les types d'énumérations :

Ces types sont créés pour définir le domaine des valeurs que peuvent prendre des objets de ce type.

Un objet de type énumération est défini par le mot-clef **enum** précédé par identificateur de type, suivis de la liste des identifiants représentant les valeurs constantes de ce type.

Les valeurs constantes peuvent être définies au cours de la déclaration ou sont codées par défaut comme suite de valeurs entières à partir de 0.

Syntaxe:

enum identificateur{

constante 1, constante 2, constante 3, ….., constante n

} variables;

Les valeurs associées aux différentes constantes symboliques sont par défaut définies de la manière suivante:

La première constante est associée à la valeur 0, les constantes suivantes suivent une progression de 1.

Exemple:

enum Jours{lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche}Jr;

lundi correspond à la valeur 0.

jeudi correspond à la valeur 3.

dimanche correspond à la valeur 6.

Exemple:

```
enum jours{ lundi, mardi, mercredi,jeudi, vendredi, samedi,diamche} ;

union date{
        enum jours j;
        int anne;
};

int main(int argc, char** argv){
        union date d;
        d.anne=2019;
        printf("l'annee :%d\n", d.anne);
        d.j=vendredi;
        printf("le numero du jour %d       l'annee: %d", d.j, d.anne);
return(0);
}
```

6- Gestion des fichiers

Objectif: Nous allons voir dans ce chapitre, comment nous pouvons créer, lire et modifier nous-mêmes des fichiers sur les périphériques externes(disque dur, clé UBS,….).

I) Définitions

a- Un Fichier :

Un **fichier (anglais: file) est un ensemble** homogène de données, ordonnée et de taille dynamique. Il fut un document que l'on pourrait garder **sur un support secondaire** (externe) (disque dur, clé USB, disque optique, bande magnétique, ...). Un **fichier structuré** contient généralement une suite d'enregistrements(struct) homogènes, qui regroupent le plus souvent plusieurs composantes appelés champs.

Pour pouvoir manipuler un fichier en C, un programme a besoin d'un certain nombre d'informations :

- l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier,
- la position de la tête de lecture/ écriture ,
- le mode d'accès au fichier (lecture ou écriture)...

Ces informations sont rassemblées dans une structure, dont le type, **FILE ***, est défini dans <stdio.h>.

Un objet de type FILE * est appelé flot de données (stream en anglais).

b- Fichier séquentiel :

Dans des ***fichiers séquentiels, les enregistrements sont mémorisés consécutivement dans l'ordre de*** leur entrée et peuvent seulement être lus dans cet ordre. Si on a besoin d'un enregistrement précis dans un fichier séquentiel, il faut lire tous les enregistrements qui le précèdent, en commençant par le premier.

Il existe, en général, deux types de fichiers :

- Les fichiers textes : sont considérés comme un ensemble de données. Avec ce type de fichier, le système réagit aux caractères de contrôle (retour à la ligne(\n), tabulation (\t),…)

- Les fichiers binaires : sont des suites d'octets qui peuvent faire l'objet, par exemple, de collections d'entités constituées d'un nombre précis d'octets (enregistrements ou structures). Dans ce cas, le système n'attribue aucune signification aux caractères de contrôle transférés depuis ou vers le périphérique.

c- Fichiers standards :

Il existe trois fichiers spéciaux qui sont définis par défaut pour tous les programmes :

- ***stdin le fichier d'entrée standard*** (par défaut, le clavier)
- ***stdout le fichier de sortie standard*** (par défaut, l'´ecran)
- ***stderr le fichier de sortie d'erreur*** (par défaut, l'´ecran)

Remarque sur la mémoire tampon :

*Les accès à un fichier se font par l'intermédiaire d'une **mémoire tampon** (angl.: buffer) qui est une zone de la mémoire centrale de la machine réservée à un ou plusieurs enregistrements du fichier. L'utilisation de la mémoire tampon a pour effet de réduire le nombre d'accès à la périphérie et implicitement et le nombre des mouvements de la tête de lecture/écriture.*

2) Ouverture et fermeture d'un fichier

a- Ouverture d'un fichier se fait par la fonction prédéfinie: **fopen**

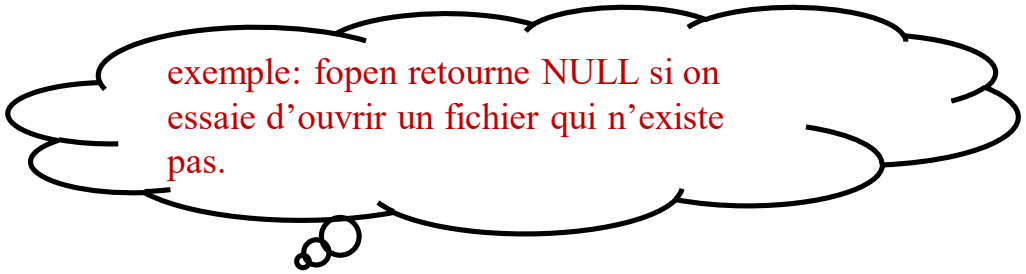
Un fichier possède deux noms: un **nom logique** (nom externe ou nom Dos de type chaîne de caractères) et **un nom physique** (nom interne) de type pointeur sur FILE. Un fichier est parcourus par son nom physique.

Syntaxe :

FILE *nom_interne ;

Exemple :

FILE *fiche;



L'ouverture d'un fichier se fait par la fonction **fopen** qui retourne un pointeur de type structure FILE ou NULL en cas d'échec d'ouverture du fichier.

Cette fonction accepte deux arguments de type chaine de caractères.

Sa syntaxe est :

fopen("chemin+nom-de-fichier","mode")

Le premier argument fournit donc le nom et le chemin du fichier . Le second argument, mode, est une chaîne de caractères qui spécifie le mode d'accès au fichier. Les spécificateurs de mode d'accès diffèrent suivant le type de fichier considéré.

On distingue:

Le paramètre **mode** permet d'indiquer le mode de fichier (binaire ou texte) et le mode d'ouverture (création, lecture, écriture ou mise à jour). Il s'agit des paramètres :

b : indique le mode binaire.

t : indique le mode texte.

w : ouvre un fichier en écriture. Il le crée s'il n'existe pas et l'écrase s'il existe déjà.

r : ouvre un fichier en lecture seule.

a : permet d'ajouter des éléments à la fin d'un fichier existant. Comme pour le cas du mode w, si le fichier n'existe pas, il est créé ; si le fichier existe déjà, son ancien contenu est conservé.

r+: permet de mettre à jour un fichier (modifier des éléments).

w+: crée un fichier avec accès en lecture et écriture.

a+ : permet de faire des ajouts en fin de fichier et des mises à jour.

"r+b":ouverture d'un fichier binaire en lecture/écriture

"w+b":ouverture d'un fichier binaire en lecture/écriture mais supprime le contenu du fichier si le fichier existe déjà

"a+b" :ouverture d'un fichier binaire en lecture/écriture à la fin

Remarques:

- *Il est conseillé donc de tester toujours si la valeur renvoyée par la fonction* ***fopen*** *est égale à NULL afin de détecter les erreurs d'ouverture de fichier (lecture d'un fichier inexistant...).*
- *Lorsqu'on ouvre un fichier, son pointeur pointe sur son premier élément.*
- *lorsqu'on ouvre un fichier sans spécifié t ou b(texte ou binaire), il est par défaut en mode texte.*

Exemple*:*

```
#include <stdio.h>
main( ) {
      File *f;
      f=fopen(" test.txt", "rt"); //ouverture du fichier texte test.txt en lecture
      if (f==NULL)   /*affiche un message s'il y a problème dans l'ouverture du fichier
                printf("Erreur");
      else{
                … /*instructions de lecture*/
                fclose (f); /*ferme le fichier*/
      }
      f=fopen("document.bat", "wb"); /*crée le fichier binaire document.bat*/
      if (f==NULL)    printf("Erreur");
      else{    …….
         fclose (f); /*ferme le fichier*/
}
}
```

La fonction fclose() ferme le fichier dont le nom interne est indiqué en paramètre. Elle retourne 0 si l'opération s'est bien déroulée, -1 en cas d'erreurs.

b- Accès pour la lecture et l'écriture au contenu d'un fichier

Une fois le fichier est ouvert, le langage c permet plusieurs types d'accès à ce fichier que ce soit en lecture ou en écriture:

- par caractère,
- par ligne,
- par enregistrement,
- par données formatées,

avec des entrées/sorties formatés ou non

3- Traitement par caractères dans un fichier séquentiel :

Similaires aux fonctions getchar et putchar, les fonctions fgetc et fputc permettent respectivement de lire et d'écrire un caractère dans un fichier.

a) Ecrire un caractère - fputc :

```
int fputc(int varCar, FILE *Flot);
```

fputc transfère le caractère indiqué par varCar **dans le fichier référencé par Flot et avance la** position de la tête de lecture/écriture au caractère suivant.

varCar représente un caractère (valeur numérique de 0 à 255) ou le symbole de fin de fichier EOF si l'indicateur atteint la fin.

Flot est un pointeur du type FILE* qui est relié au nom du fichier (nom externe).

Remarque :

L'instruction **fputc('a', stdout);**

est identique à **putchar('a');**

b) Lire un caractère - fgetc :

int varCar= **f**getc(FILE* Flot);

fgetc fournit comme résultat le prochain caractère du fichier référencé par Flot **et avance la** position de la tête de lecture/écriture au caractère suivant. A la fin du fichier, **fgetc retourne EOF**.

varCar représente une variable du type int qui peut accepter une valeur numérique de 0 à 255 ou le symbole de fin de fichier **EOF.**

Flot **est un pointeur du type FILE* qui est relié au nom du fichier à lire (nom externe).**

Remarque :

L'instruction **C = fgetc(stdin);**

est identique à **C = getchar();**

Exemple:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define ENTREE "entree.txt"
int main(int argc, char** argv) {
      FILE *f_in ;    int c;
      if ((f_in = fopen(ENTREE,"w")) == NULL) {// Ouverture du fichier ENTREE en écriture
            printf("\nErreur: Impossible de lire %s\n",ENTREE);  return(0);
      }
      for(int i=0;i<6;i++){
            c=getchar(); fflush(stdin);
            fputc(c,f_in);
      }
      fclose(f_in);// Fermeture des flots de données
      if ((f_in = fopen(ENTREE,"r")) == NULL) {// Ouverture du fichier ENTREE en lecture
            printf("\nErreur: Impossible d'écrire dans %s\n",ENTREE); return(0);
      }
      while ((c = fgetc(f_in)) != EOF){
            fputc(c, stdout);   printf(" , ");  }
      fclose(f_in);// Fermeture des flots de données
return(0);
}
```

4- Lecture et écriture par lignes sur fichier

a) Lecture par ligne : fgets

char *fgets (char *chaine , int taille , FILE *Flot);

- **chaine** est de type pointeur vers char et doit pointer vers un tableau de caractères.
- **taille** est la taille en octets du tableau de caractères pointé par chaîne (au maximum la taille de la zone de stockage réservée à la variable **chaine**),.
- **Flot** est de type pointeur vers FILE. Il pointe vers le fichier à partir duquel se fait la lecture.

La fonction fgets retourne le pointeur chaine cas de lecture sans erreur, ou **NULL** dans le cas de **fin de fichier** ou d'erreur et **non pas EOF**.

Remarque:

La fonction fgets lit les caractères (taille-1) du fichier et les range dans le tableau pointé par chaine.

b) Ecriture par lignes : fputs

int fputs (char *chaine , FILE *Flot);

- **chaine** est de type pointeur vers char. Pointe vers un tableau de caractères contenant une chaine se terminant par un NULL.
- **Flot** est de type pointeur vers FILE. Il pointe vers le fichier sur lequel se fait l'écriture.

La fonction fputs rend une valeur non négative si l'écriture se passe sans erreur, et EOF en cas d'erreur(de valeur -1).

Remarques:

La fonction puts et gets sont équivalentes aux fonctions fputs(chaine, stdout) et fgets(chaine, taille, stdin) respectivement.

L'affichage sur la sortie standard avec **puts** permet le retour à la ligne au contraire avec l'appel de **fputs.**

Exemple:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define ENTREE "entree.txt"
int main(int argc, char** argv) {
    FILE *f_in;    int c ; char *s=(char*)malloc(200);
    if ((f_in = fopen(ENTREE,"w")) == NULL) {// Ouverture du fichier ENTREE en écriture
        printf("\nErreur: Impossible de lire %s\n",ENTREE);  return(0);
    }
    for(int i=0;i<6;i++){
        gets(s); fflush(stdin);
        fputs(s,f_in);
    }
    fclose(f_in);// Fermeture des flots de donnees
    printf("\n ");
    if ((f_in = fopen(ENTREE,"r")) == NULL) {// Ouverture du fichier ENTREE en lecture
        printf("\nErreur: Impossible d'ecrire dans %s\n",ENTREE); return(0);
    }
    while ((s = fgets(s,10,f_in)) != NULL){
        fputs(s, stdout); printf(" , ");  }
    fclose(f_in);// Fermeture des flots de données
return(0);
}
```

On écrit 6 chaines de caractères dans le fichier

La lecture se fait par 10 par 10 caractères

5- Lecture / écriture en mode binaire dans un fichier (fread et fwrite)

Les fonctions d'entrées-sorties binaires permettent d'écrire des données dans un fichier sans transcodage. Du cous, ces fonction sont portables et efficaces par rapport aux fonctions standards d'entrée sortie. Elles écrivent ce qu'on leur dit.

Elles sont notamment utiles pour manipuler des données de grande taille et surtout les tableaux et les structures complexes.

Principales fonctions (fichiers binaires) :

int fwrite (type *bloc, int taille, int nbr, FILE *Flot);

Ecrit **nbr** éléments de taille **taille** (octets) à partir de l'emplacement mémoire pointé par **bloc** dans le fichier **Flot**. Rend le nombre de blocs effectivement copiés (écrits).

int fread (type *bloc, int taille, int nbr, FILE *Flot);

Lit **nbr** éléments de taille **taille** (octets) dans le fichier **Flot**. Le résultat est stocké à l'emplacement pointé par **bloc**. Rend le nombre d'éléments lus .

Exemple d'écriture dans un fichier

```
#include <stdio.h>
typedef struct{
      char Nom [6], Prenom [6];
}personne;
main( ){
      personne p;
      int n=0;
      FILE *f=fopen("personnes.bat", "wb");
      if (f==NULL)    printf("Erreur d'ouverture");
      else  {
                printf("entrer une personne:\n");
                scanf("%s %s",p.Nom,p.Prenom);
                n=fwrite(&p,12,1,f);
                if (n != 1)    printf("Erreur d'écriture");
      fclose (f);
      }
}
```

- **Exemple de Lecture de fichiers**

```
#include <stdio.h>
typedef  struct   {
      char Nom [6], Prenom [6];
}personne;
main( ) {
      FILE *f;
      personne p; /*p variable de type personne*/
      int n;
      f=fopen("personnes.bat", "rb"); /*ouvre le fichier personnes.bat en lecture*/
      if (f==NULL)
            printf("Erreur d'ouverture");
      else {
            while (n=fread(&p,12,1,f))
                  printf("%s %s",p.Nom,p.Prenom);
      fclose (f);
      }
}
```

Exemple 3:

```
#include <stdio.h>
#include <stdlib.h>
#define NB 50
#define F_SORTIE "sortie.txt"
struct personne{
        char nom[6],  pren[6];
};
void afficheStruct(struct personne p);
struct personne lecture();
void lectFichier(char *nomf);
int main(int argc, char** argv){
        FILE *f_in, *f_out;  char *ss;       struct personne p;
   if ((f_out = fopen(F_SORTIE, "w")) == NULL){
                fprintf(stderr, "\nImpossible d'ecrire dans %s\n",F_SORTIE);
                return(EXIT_FAILURE);
        }
        for(int i=0;i<3;i++){
                p=lecture(); fflush(stdin);
                fwrite(&p, sizeof(struct personne), 1, f_out);    }
        fclose(f_out);
        ss= (char*)malloc(200);
        ss="affichage du contenu du ficjhier :";
        lectFichier(ss);
}
```

```
struct personne lecture(){
	struct personne p;
	printf("donner le nom :"); scanf("%s",p.nom);
	printf("donner le prenom :"); scanf("%s",p.pren);
	return p;
}
void afficheStruct(struct personne p){
	printf(" le nom :%s  et le prénom: %s\n",p.nom,p.pren);
}
void lectFichier(char *nomf){
	FILE *f_in; int i;
	struct personne p;
	printf("\n %s\n",nomf);
	if ((f_in = fopen(F_SORTIE, "r")) == NULL) {
		fprintf(stderr, "\nImpossible de lire dans %s\n",F_SORTIE);
	}
	else{
	while(!feof(f_in)){
		if(fread(&p,sizeof(struct personne),1,f_in)!=0)
		    afficheStruct(p);
	}
}}
```

6- Lecture non séquentielle d'un fichier : fseek, rewind et ftell

Les différentes fonctions d'entrées-sorties permettent d'accéder à un fichier en mode séquentiel : les données du fichier sont lues ou écrites les unes à la suite des autres.

Il est également possible d'accéder à un fichier en mode direct, c'est-à-dire que l'on peut se positionner à n'importe quel endroit du fichier, remettre le pointeur au début et savoir sa position courante.

- La fonction **fseek** permet de se positionner à un endroit précis.

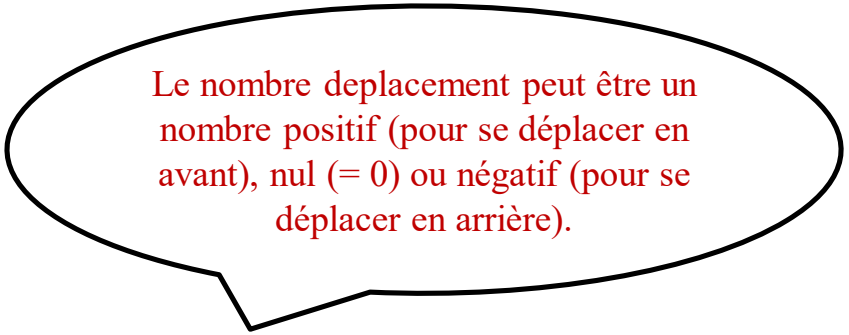
```
int fseek( FILE *flot, long deplacement, int origine);
```

La variable **deplacement** détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à **origine(fin , début du fichier ou position courante)**, compté en nombre d'octets.

Elle retourne zéro en cas de succès et une autre valeur en cas d'échec.

La variable origine peut prendre trois valeurs :

1. SEEK_SET (égale à 0) : début du fichier ;
2. SEEK_CUR (égale à 1) : position courante ;
3. SEEK_END (égale à 2) : fin du fichier.



Exemple:

```
#include <stdio.h>
#include <stdlib.h>
#define F_SORTIE "sortie.txt"
struct personne{
        char nom[6],  pren[6];
};
int main(int argc, char** argv){
        FILE *f_out;            char *ss; int pos;
        struct personne p;   int n;
    printf("entrer une  position sur le fichier : ");
    scanf("%d",&pos);
        if ((f_out = fopen(F_SORTIE, "r")) == NULL) {
                fprintf(stderr, "\nImpossible de lire dans %s\n",F_SORTIE);
        }
        else {
                if (fseek (f_out,pos*sizeof(personne),0)==0 ){
                                fread(&p, sizeof(personne),1,f_out);
                                printf(" le nom :%s  et le prénom: %s\n",p.nom,p.pren);
                }
                else            printf("Erreur d'accès");
        fclose (f_out);
} }
```

- La fonction rewind remet le pointeur au début de fichier :

rewind (FILE *flot);

- La fonction ftell retourne la position actuelle de type long (exprimée en octets) par rapport au début du fichier.

ftell (FILE *flot);

Exemple:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#define NB 25
#define F_SORTIE "sortie.txt«

void ecrireFichier(int *);
void AffichageTab(int *tt);
int *lectureFichier();
void positionSurfichier(int pos);
```

```
int main(int argc, char** argv){
        FILE *f_in;       int *tab,  i=0;
        tab = (int*)malloc(NB * sizeof(int));
        for (i = 0 ; i < NB; i++)   tab[i] = i*i;
        ecrireFichier(tab);
        AffichageTab(lectureFichier());
        if ((f_in = fopen(F_SORTIE, "r")) == NULL) {fprintf(stderr, "\nImpossible de lire dans %s\n",F_SORTIE);     return(0);
        }
        fseek(f_in, -10*sizeof(int),SEEK_END );  /* place le curseur de 10 éléments / à la fin du fichier*/
        fread(&i, sizeof(int), 1, f_in);
        printf("\n\n i = %d: la valeur lue après déplacement en arrière par 10 / a la fin du fichier, ftell(f_in)= %ld en octets\n",
        i,ftell(f_in));
        fseek(f_in, 5 * sizeof(int),SEEK_SET);  /* place le curseur de 5 éléments / au début du fichier */
        fread(&i, sizeof(int), 1, f_in);
        printf("i = %d;: la valeur après déplacement en avant 5 / au début du fichier, ftell(f_in)= %ld en octets\n", i,ftell(f_in));
        fseek(f_in, 0 , SEEK_END);              /* on se positionne a la fin du fichier */
        fseek(f_in, -1*sizeof(int), SEEK_CUR);  /* on se positionne sur le dernier élément du fichier */
        fread(&i, sizeof(int), 1, f_in);
        printf(" i = %d : la valeur lue après un déplacement en arrière / à la position courante, ftell(f_in)= %ld en octets \n",
        i,ftell(f_in));
        rewind(f_in);                           /* retour au début du fichier */
        fread(&i, sizeof(i), 1, f_in);
        printf(" i = %d: la valeur lue après retour au début de fichier avec rewind, ftell(f_in)= %ld en octets\n", i,ftell(f_in));
        fclose(f_in);
return(0);
}
```

```
void positionSurfichier(int pos){
	FILE *f_in; int i;
	if ((f_in = fopen(F_SORTIE, "r")) == NULL) {
		fprintf(stderr, "\nImpossible de lire dans %s\n",F_SORTIE);
	}
	fseek(f_in, pos*sizeof(int), SEEK_END);		/* on se positionne a la fin du fichier */
	fread(&i, sizeof(int), 1, f_in);
	printf("\t i = %d  sa position a la fin %ldoctets\n", i,ftell(f_in));
	fclose(f_in);
}
void AffichageTab(int *tt){
	printf(" \n le contenu du tableau:\n");
		for (int i = 0 ; i < NB; i++) printf("  %d",*(tt+i));
}
void ecrireFichier(int *t){
	FILE *f_in;
	if ((f_in = fopen(F_SORTIE, "w")) == NULL){
		fprintf(stderr, "\nImpossible d'ecrire dans %s\n",F_SORTIE);
	}
	fwrite(t, NB * sizeof(int), 1, f_in);
	fclose(f_in);
}
int *lectureFichier(){
	int *tt= (int*)malloc(NB * sizeof(int));	FILE *f_out;
	if ((f_out = fopen(F_SORTIE, "r")) == NULL) {
		fprintf(stderr, "\nImpossible de lire dans %s\n",F_SORTIE);	return(0);
	}
	fread(tt, NB*sizeof(int), 1, f_out);
	fclose(f_out);
	return tt;
}
```

7- Les entrées-sorties formatées dans un fichier

a) La fonction d'écriture en fichier fprintf

La fonction fprintf, analogue à printf, permet d'écrire des données dans un flot. Sa syntaxe est :

```
int fprintf(FILE *Flot, " format", expression1, . . . , expressionn);
```

Où Flot est le flot de données retourné par la fonction fopen.

format est toujours une chaine de caractères dans laquelle on insert des éléments spéciaux(%s, %d, %f, etc…) les mêmes que ce qui manipulés avec printf.

```
fprintf(stdout, " format", expression1, . . . , expressionn); <==> printf(" format", expression1, . . . , expressionn);
```

La fonction retourne 1 si l'écriture s'est bien passée et -1 sinon.

b) La fonction de saisie en fichier fscanf

La fonction fscanf, analogue à scanf, permet de lire des données dans un fichier. Sa syntaxe est semblable à celle de scanf :

int fscanf(FILE *Flot, "format", arg1, . . . , argn);

où Flot est le flot de données retourné par fopen.

format est toujours une chaine de caractères dans laquelle on insert des éléments spéciaux(%s, %d, %f, etc…) les mêmes que ce qui manipulés avec scanf.

fscanf(stdin, " format", expression1, . . . , expressionn); <==> scantf(" format", expression1, . . . , expressionn);

La fonction retourne 1 si la lecture s'est bien passée et -1 sinon.

Exemple:

```
void lectureFich(char *nomf);
void affichageFich(char *nomf);
int main(int argc, char** argv) {
        char *nomf=(char*)malloc(20);
        printf("donneer le nom et le chemein du fichier :");       scanf("%s",nomf);
        lectureFich(nomf);
        affichageFich(nomf);
return 0;
}
void lectureFich(char *nomf){
        FILE *fin;
        char *nom[20]; float note;
        fin=fopen(nomf,"w");
        if(fin==NULL)
                printf("problème d'ouverture dun fichier !!!!");
        else {
           for(int i=0;i<3;i++){fflush(stdin);
                printf("donner le nom :"); scanf("%s",nom);
                printf("donner la note :"); scanf("%f",&note);
                fprintf(fin,"%s %f",nom,note);
             }
        }
        fclose(fin);
}
```

```
void affichageFich(char *nomf){
   FILE *fin;
   char nom[20]; float note;
   if((fin=fopen(nomf,"r"))==NULL)
   printf("problème d'ouverture dun fichier !!!!");
   else {
     while(!feof(fin)){
       fscanf(fin,"%s %f",nom,&note);
       printf("-- le nom :%s",nom);
       printf("   la note :%f\n",note);
    }
}
fclose(fin);
}
```

8- Renommer et supprimer un fichier

- **Renomme un fichier**

int rename(const *char** ancienNom, const *char** nouveauNom);

- **supprimer un fichier**

int remove(const char* fichierASupprimer);

Les fonctions renvoient 0 si elles ont réussi à renommer ou supprimer le fichier , sinon elles renvoient une valeur différente de 0.

Attention! **La fonction remove supprime le fichier indiqué sans demander de confirmation ! Le fichier n'est pas mis dans la corbeille et peut ne pas être récupérable.**

Exemple:

```
if(rename("c://test/test.txt", "c://test/fich.txt")==0)
        printf("l'opération c'est bien déroulé!!!\n");
else printf("problème pour renommer le fichier \n");
if(remove("c://test/test.txt")==0)
        printf("la suppression  c'est bien déroulé!!!\n");
else  printf("problème pour supprimer  le fichier \n");
```

9- Lister le contenu d'un dossier

Il est possible de lister le contenu d'un dossier (fichiers et sous-dossiers) sous Windows, on utilise les fonctions FindFirstFile() et FindNextFile() "*.*" .

Le HANDLE(la référence ou bien l'indice) retourné par FindFirstFile() se refert à la liste des fichiers et les sous-dossiers trouvés et doit être fermé à la fin avec la fonction FindClose(). Il existe d'autre fonctions pour lister le contenu du répertoire courant.

Exemple:

```
//#include <windows.h>
int main(int argc, char** argv){
   WIN32_FIND_DATA File;
   HANDLE hsch;
   hsch = FindFirstFile("c://*.*", &File); //affiche tout le contenu du dossier racine c:
   if (hsch != INVALID_HANDLE_VALUE)     {
      do {
         printf("%s\n", File.cFileName);
      } while (FindNextFile(hsch, &File));

      FindClose(hsch);
   }
return(0);
}
```

Dans un environnement conforme à la norme POSIX, une norme qui regroupe un ensemble de fonctions permettant de constiter les services de base d'un système d'exploitation, on utilisera plutôt opendir(), readdir() puis closedir().

Exemple:

```
#include<dirent.h>
int main(int argc, char** argv){
    DIR * rep = opendir("c://test/.");
    char **s;    int j=0;
    s=(char **)malloc(50*sizeof(char*));
    for(int i=0;i<50;i++)    *(s+i)=(char *)malloc(30*sizeof(char));
   if (rep != NULL)    {
       struct dirent * ent;
       while ((ent = readdir(rep)) != NULL)      {
           strcpy(*(s+j),ent->d_name);  j++;
       }
       closedir(rep);
   }
   printf("le nombre de fichiers et sous dossierts %d\n",j);
   for(int i=0;i<j;i++) printf("%s\n", *(s+i));
 return(0);
}
```

Fin