Université Moulay Ismail
Faculté des Sciences et Technique
Département d'Informatique

Structures de données en C

Filière MIP, Module I143, S4

Préparé et présenté par:
Pr. B. BOUDA

Année Universitaire 2019/2020

Objectifs du module:

→Maitriser les structures de données élémentaires: les structures, les listes chaînées, les piles, les files et les arbres.
→Utiliser les concepts des structures de données élémentaires pour résoudre quelques problèmes simples.

Pré-requis du module:

→Avoir de bonnes connaissances en algorithmique et programmation en C.

Chapitre 1
Rappels et compléments de C

- Partie I: Les rappels
- Partie II: Les structures
- Partie III: La gestion de fichiers

Partie I:
Les rappels

Introduction

- D'une manière générale, l'emploi de **données statiques** présente certains défauts. Par exemple, ils ne permettent pas de définir des tableaux de dimensions variables. En effet, on fixe préalablement la taille des tableaux (100 octets par exemple). Par conséquence, une mauvaise utilisation de l'ensemble de la mémoire se produit.
- L'emploi de **données dynamiques** va permettre de pallier le défaut dû à l'emploi de **données statiques,** en donnant au programmeur l'opportunité d'allouer et de libérer librement la mémoire (c'est la gestion dynamique de la mémoire).
- Les fonctions telles que **malloc, calloc**, **realloc** et **free** de la bibliothèque standard (**stdlib.h**) permettent la gestion de la mémoire dynamiquement.

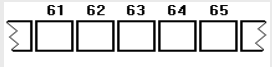
Variables

- Une variable est une **zone mémoire** dans laquelle on peut mémoriser d'une façon **temporaire** une valeur pour une exploitation ultérieur.

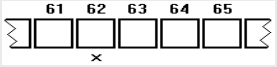
Exemple I-1

soit x une variable:

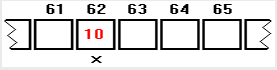
cases mémoire: // Les emplacements de la mémoire



int x; // Déclaration: instruction qui réserve un emplacement de mémoire pour un entier x



x = 10; // Initialisation: instruction qui écrit la valeur 10 dans l'emplacement réservé par x



Variables

- La taille d'une variable dépend de son type:
  - char : 1 octet
  - int : 2, 4 ou 8 octets (selon l'architecture du système)
  - float : 4 octets
  - double : 8 octets
  - etc
- L'opérateur «sizeof()» donne la taille en octets du type ou de la variable passée en paramètre.

Exemple I-1 suite

```
void main(){
printf("Sur mon système un char fait %d octets", sizeof(char));
}
```

- **Affichage:**

Sur mon système un char fait 1 octets

Variables tableaux

- On appelle tableau **une variable** composée de données de **même type**, stockées de manière **contiguë** en mémoire (les unes à la suite des autres).

Exemple I-1 suite

Éléments du tableau	2	1	8	5	2	3
Indices du tableau	0	1	2	3	4	5

- La déclaration d'un tableau s'effectue en précisant le **type** de ses éléments, son **nom** et sa **dimension**.

Exemple I-1 suite

```
int t[8];  /*déclaration du tableau t de 8 entiers */
//on peut effectuer la déclaration et l'initialisation au même temps:
int t[8] = {0, 1, 2, 3, 4, 5, 6, 7 }; /*chaque case prend une valeur*/
int t[ ] = {1, 2, 3, 4, 5, 6, 7, 8 };  /* le compilateur compte à votre place */
int t[8] = {1, 2, 3, 4};               /* 8 places prises, 4 initialisées*/
```

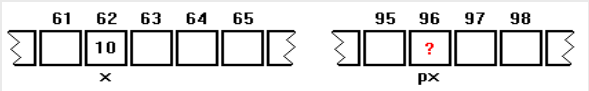
Pointeurs

■ Un pointeur est une variable **spéciale** qui peut contenir l'**adresse** d'une autre variable.

Exemple I-2

soit x un entier et px un pointeur:

```
int* px; /*Déclaration: instruction qui réserve un emplacement pour stocker une
           adresse mémoire*/
```




```
px = NULL; /* Initialisation: Identique à l'initialisation à 0 d'une variable entière */
px = &x; /*Initialisation: instruction qui écrit l'adresse de x dans l'emplacement
           réservé par px*/
```

Remarque

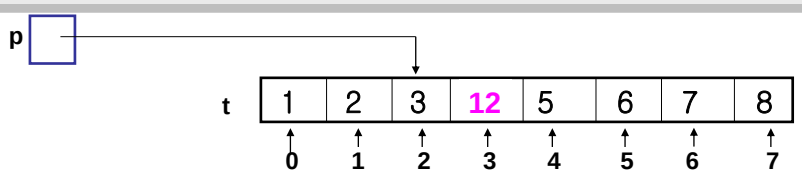
❑ La taille d'un pointeur est de 4 octets (quel que soit le type sur lequel il pointe): sizeof(char*)=sizeof(int*)=sizeof(double*)=4

Tableaux et pointeurs

■ Les **tableaux** et les **pointeurs** sont de même type, on peut donc utiliser la notation «[ ]» à partir des variables de types pointeurs. Voici un exemple:

Exemple I-2 suite

```
void main(){
  int  t[8]={1,2,3,4,5,6,7,8};        /* t est un tableau de 8 entiers*/
  int * p;          /* p est un pointeur sur un entier */
  p = &t[2];      /* p pointe sur la 3ème case du tableau (t[2]=3)*/
  p[1] = 12; (ou t[3] = 12;)  /*on écrit 12 dans la 4ème case du tableau */
}
```

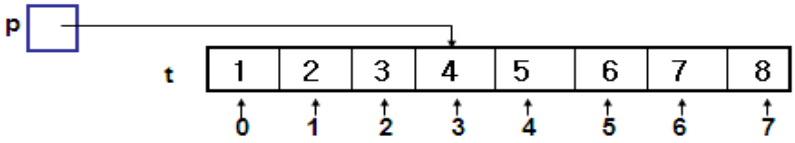


Opérateurs sur les pointeurs

■ Tous les opérateurs (arithmétiques, d'affectation, d'affectation combinée, d'incrémentation de décrémentation,…) d'entiers existent également pour les pointeurs: «++», «--», «=», «+=», «-=», …

Exemple I-2 suite

```
void main(){
  int  t[8]={1,2,3,4,5,6,7,8};      /* t est un tableau de 8 entiers*/
  int * p;        /* p est un pointeur sur un entier */
  p = &t[2];     /* p pointe sur la 3ème case du tableau (t[2]=3)*/
  p=p+1; (ou p=p++;) /*incrémentation: p pointe sur la 4ème case du tableau*/
}
```



Arithmétique des pointeurs

■ La déclaration d'un **tableau** réserve la place en mémoire pour les éléments du tableau et fournit une **constante** de **type pointeur** (identifiée par le même nom donné au tableau) qui contient l'adresse du premier élément du tableau.

Exemple I-2 (suite)

```
void main(){
 int  t[8];          /* Déclaration d'un tableau t de 8 entiers */
 t + 0 = &t[0]  /* t contient l'adresse du 1er élément du tableau */
 t + 1 = &t[1]  /* t +1 contient l'adresse du 2ème élément du tableau */
 t + 2 = &t[2]  /* t +2 contient l'adresse du 3ème élément du tableau */
 ….
 t + i = &t[i]     /* t +i  contient l'adresse du ième élément du tableau */
}
```

Allocation statique

■ La réservation de la mémoire peut se faire **statiquement:** au moment de la compilation d'un pgm. Voyez l'exemple suivant:

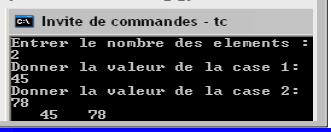
Exemple I-3

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
….
void main( ){
int i, taille;

/*Allocation d'un bloc de 50 éléments*/
int tab[50];
printf("Entrer le nombre des
élements :\n");
scanf("%d", &taille);
```

Exemple I-3 (suite…)

```
/* Saisie du tableau */
for(i=0; i <taille; i++){
  printf("Donner la valeur de la
  case %d: \n", i+1);
  scanf("%d", &tab[i]);
}
/* Affichage du tableau */
for(i=0; i<taille; i++){
    printf("%5d", tab[i]);
}
}
```



Allocation dynamique

■ La réservation de la mémoire peut se faire **dynamiquement**: pendant l'exécution du pgm. Voyez l'exemple ci-dessous:

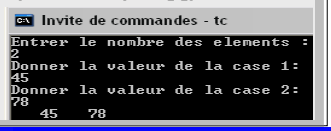
Exemple I-3bis

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
void main( ){
int i;
int taille, *ptT;
printf("Entrer le nombre d'eles :\n");
scanf("%d", &taille);

/*Allocation d'un bloc de 50 élements/
ptT= (int*)malloc(50);
```

Exemple I-3bis (suite…)

```
/* Saisie du tableau */
for(i=0; i <taille; i++){
  printf("\nDonner la valeur de
  la  case %d: \n", i+1);
  scanf("%d", &ptT[i]);
}
/* Affichage du tableau */
printf("Tab final:\n");
for(i=0; i<taille; i++){
    printf("%5d", ptT[i]);
}
}
```



La fonction malloc()

- L'instruction **ptT=(int*) malloc(50);** alloue un bloc de **50 octets** dans la mémoire et renvoie le pointeur **ptT** sur le début de la zone allouée ou **NULL** si l'allocation échoue.
- L'instruction **for** qui vient à la suite de **malloc()** n'est donnée qu'à titre d'exemple d'utilisation de la zone ainsi créée !

Remarque

- Afin d'assurer une portabilité au programme, on emploi l'opérateur **sizeof(type)** et **taille** (taille de la zone allouée) dans l'argument de **malloc** !
- Afin d'éviter les erreurs du bug, on utilise l'allocation avec conversion par l'emploi de **(type*)** avant **malloc()** **!**

- Voici le prototype de **malloc()** (qui figure dans stdlib.h):

Prototype

```
void * malloc (size_t taille);                    (stdlib.h)
//taille c'est la taille de la zone allouée et size_t est le nom de son type prédéfini
```


La fonction free()

- Le rôle de la fonction **free()** est de libérer un emplacement mémoire préalablement alloué. Voyez le prototype de **free()**:

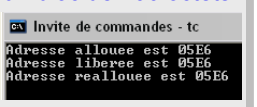
Prototype

```
void free( void *ptr);                            (stdlib.h)
//ptr est un pointeur sur le bloc préalablement alloué
```

- L'exemple suivant nous montre comment **malloc()** peut profiter d'un espace préalablement libéré par **free()**:

Exemple I-4

```
void main(){
int n=100;
char *ptr1, *ptr2 ;
ptr1= (char*) malloc(n*sizeof(char)); /*allocation  d'un bloc de 100 octets
printf ("Adresse allouée est %p\n", ptr1);
free(ptr1);/*libération du bloc préalablement alloué*/
printf ("Adresse libérée est %p\n", ptr1);
ptr2= (char*) malloc(n*sizeof(char)); /*réallocation  d'un bloc de 100 octets
printf ("Adresse réallouée est %p\n", ptr2);}
```



La fonction calloc()

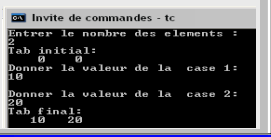
■ Contrairement à ce qui se passait avec **malloc()**, **calloc()** permet de réserver un certain nombre d'octets et de les **initialiser à zéro**.

Exemple I-5

```
#include <stdio.h>
#include <stdlib.h>
void main( ){
int i, taille, *ptT;
printf("Entrer le nombre d'elem :\n");
scanf("%d", &taille);
/*Allocation d'un bloc d'elem*/
ptT=(int*)calloc(taille,sizeof(int));
/*Affichage du tableau initial*/
printf("Tab initial: \n");
for(i=0; i<taille; i=i+1){
    printf("%5d",ptT[i]);
}
```

Exemple I-5 (suite…)

```
/* Saisie du tableau */
for(i=0; i <taille; i++){
  printf("\nDonner la valeur de
  la  case %d: \n", i+1);
  scanf("%d", &ptT[i]);
}
/* Affichage du tableau final*/
printf("Tab final:\n");
for(i=0; i<taille; i++){
    printf("%5d", ptT[i]);
}
free(ptT);
}
```



La fonction calloc()

■ Prototype de la fonction **calloc()**

Prototype

void * **calloc** (size_t **nb_blocs**, size_t **taille**); (stdlib.h)
//alloue l'emplacement nécessaire à **nb_blocs** consécutifs, ayant chacun une taille de **taille** octets.

■ Contrairement à ce qui se passait avec **malloc**, où le contenu de l'espace mémoire alloué était imprévisible, **calloc** remet à zéro chacun des octets de la zone ainsi allouée.

■ La taille de chaque bloc, ainsi que leur nombre sont tous deux de type **size_t**.

■ On voit qu'il est possible d'allouer en une seule fois une place mémoire (de plusieurs blocs) beaucoup plus importante que celle allouée par **malloc** (la taille limite théorique étant maintenant **size_t*size_t** au lieu de **size_t**).

La fonction realloc()

■ La fonction **realloc()** permet de modifier la taille d'une zone mémoire préalablement allouée par **malloc()**, **calloc()** ou **realloc()**. Voyez le prototype de **realloc()**:

Prototype

```
void * realloc (void * pointeur, size_t taille);          (stdlib.h)
//pointeur doit être l'adresse de début de la zone dont on veut modifier la taille.
//taille représente la nouvelle taille souhaitée.
```

■ Exemple :

- Nous allouons un tableau de nombre (entiers) de taille désirée (nous utilisons **malloc()**);
- Nous augmentons la taille du tableau afin de retenir deux nombres supplémentaires (nous utilisons donc **realloc()**);
- Nous ajoutons les deux nombres comme «avant dernier élément» et «dernier élément du tableau».

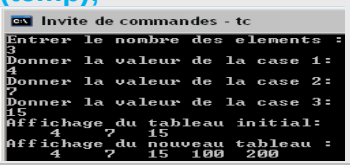
La fonction realloc()

Exemple I-6

```
#include <stdio.h>…
void main( ){
int i, taille, *ptT, *temp;
printf("Entrer le nombre des
elements :\n");
scanf("%d", &taille);
ptT=(int*)malloc(taille * sizeof(int));
for(i=0; i <taille; i++) {
   printf("Donner la valeur de la
   case %d: \n", i+1);
   scanf("%d",&ptT[i]);
}
printf("Affichage du tableau initial:\n");
for(i=0; i<taille; i++){
    printf("%5d",ptT[i]);
}
```

Exemple I-6 (suite)

```
temp=(int*) realloc(ptT,
(taille+2) * sizeof(int));
temp[taille]=100;
temp[taille+1]=200;
printf("\nAffichage du nouveau
          tableau :\n");
for(i=0; i<taille+2; i++){
    printf("%5d", temp[i]);
}
free(ptT);
free(temp);
}
```



Test d'allocation mémoire

- Il est important de vérifier si l'allocation mémoire s'est bien passée sinon, votre programme ne fonctionnera pas correctement.

➢ Il suffit de tester la valeur du pointeur après l'allocation.

Exemple I-7 :

```
void main( ){
   ptT=(int*) malloc (taille * sizeof(int));
   if(ptT==NULL){
      printf("Allocation mémoire est échouée !\n");
   }
    else{
      printf("Allocation mémoire s'est bien passée");
   }
}
```

Remarque

C'est pareil pour les fonctions **calloc ()** et **realloc()** !

En résumé

- La **mémoire** est une composante essentielle dans la programmation. De ce fait, il est intéressant de l'utiliser avec précaution.

- Nous avons découvert que la gestion de la mémoire d'une façon **dynamique** permet d'allouer et de libérer de la mémoire avec flexibilité. Par conséquent, de pallier le défaut dû à l'emploi de données **statiques**.

- Il existe des fonctions prédéfinies telles que **malloc()**, **calloc()**, **realloc()** et **free()** qui permettent la gestion dynamique de la mémoire.

Partie II:
Les structures

Les structures

- **Introduction**
- Définition et déclaration d'une structure
- Manipulation des structures

Introduction

- Une **variable simple** permet de désigner sous un **nom** une **seule valeur** pour une exploitation ultérieur.
- Une **variable tableau**, permet de désigner sous un **seul nom** un **ensemble de valeurs** mais de **même type**, chacune d'entre elles étant repérée par un **indice**.
- Une **structure**, quant à elle, permet de désigner sous **un seul nom** un **ensemble de valeurs** pouvant être de **types différents**. L'accès à chaque valeur de la structure se fera, cette fois, non plus par un indice, mais par son **nom**.
- Une **structure** est une agrégation de **plusieurs valeurs** (**champs**) de types **arbitraires**.

Définir une structure

- Si on veut identifier un **étudiant** d'une classe en C, on peut le faire avec une **structure**. Voyez d'abord cette définition :

Exemple I-8

```
struct EtudiantRepere{
  char nom[30];
  char prenom[30];
  int age, note;
}; //N'oublier pas le point virgule ici !
```

- Une structure est définie par le mot clé **struct**, suivi du nom de la structure (**EtudiantRepere**) et de la liste de ses champs typés (**nom**, **prenom**, **age** et **note**), entre accolades.

Remarque

D'habitude on met la **définition de la structure** (modèle de structure) en variable globale c'est-à-dire avant le **main**.

Déclarer une structure

- Dans le corps la fonction **main()**, on déclare des **variables structures** (ou plus couramment les **structures**), voyez l'exemple suivant:

Exemple I-8 (suite) :

```
void main(){
struct EtudiantRepere Etudiant; //variable de type EtudiantRepere
}
```

Remarque

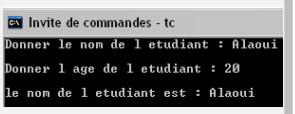
Cette déclaration réserve un emplacement pour **un seul Etudiant** de type **EtudiantRepere** (destiné à contenir exactement deux tableaux de caractères (**nom**, **prenom**) et deux entiers (**age**, **note**).

Utiliser une structure

- En C, au cours de sa déclaration on peut utiliser une structure en travaillant individuellement sur chacun de ses champs.
- Pour désigner un champ on procède dans l'ordre suivant:
  1. **nom de la variable structure,**
  2. **l'opérateur «point»,**
  3. **nom du champ désiré.**

Exemple I-8 (suite) :

```
void main(){
   struct EtudiantRepere Etudiant;
   printf("Donner le nom de l'étudiant : ");
   scanf("%s", Etudiant.nom);
   printf ("Donner l'âge de l'étudiant : ");
   scanf("%d", &Etudiant.age);
   //Affichage
   printf("\nle nom de l'étudiant est : %s", Etudiant.nom);
}
```



Regrouper la définition et la déclaration d'une structure

- Il est **recommandé** de regrouper la **définition de la structure** et la déclaration des **variables structures** dans le modèle de structure comme dans l'exemple suivant :

Exemple I-9 :

```
struct EtudiantRepere{
  char nom[30];
  char prenom[30];
  int age, note;
}Etudiant; /*déclaration de 1 variable de type EtudiantRepere*/
struct EtudiantRepere Etudiant; //une autre façon de déclaration
```

Exemple I-9 (suite) :

```
void main(){
   printf("Donner le nom de l'étudiant : ");
   scanf("%s", Etudiant.nom);
   printf("le nom de l'étudiant est : %s", Etudiant.nom);
}
```

Initialiser une structure

- Il est possible d'**initialiser** explicitement une structure lors de sa déclaration. Voici un exemple :

Exemple I-10 :

```
struct EtudiantRepere{
  char nom[30];
  char prenom[30];
  int age, note;
}Etudiant={"Alaoui","Ahmed",22,14}; //déclaration et initialisation
struct EtudiantRepere Etudiant={"Alaoui","Ahmed",22,14}; // une autre façon
```

Exemple I-10 :

```
void main(){
 printf("le nom de l'étudiant est : %s", Etudiant.nom);
 printf("\nle prenom de l'étudiant est : %s", Etudiant.prenom);
 printf("\nl age de l'étudiant est : %d", Etudiant.age);
 printf("\nla note de l'étudiant est : %d", Etudiant.note);
}
```

```
Invite de commandes - tc
le nom de l etudiant est : Alaoui
le prenom de l etudiant est : Ahmed
l age de l etudiant est : 22
la note de l etudiant est : 14
```


Tableaux de structures

- Imaginez que la structure **Etudiant** représente cette fois un tableau de **40 étudiants**. Voyez cet exemple:

Exemple I-11 :

```
#define taille 40  //ici on définit un tableau de 40 éléments
struct EtudiantRepere{
  char nom[30], prenom[30];
  int age, note;
}Etudiant[taille]; //déclaration de 1 tableau de structure de type EtudiantRepere
struct EtudiantRepere Etudiant[taille]; //une autre façon de déclaration
```

Exemple I-11 (suite):

```
void main(){
 int i;
 for(i=0; i<taille; i++){
    printf("Donner le nom de l'étudiant %d : ", i+1);
    scanf("%s", Etudiant[i].nom); }
 for(i=0; i<taille; i++){
    printf("\nLe nom de l'étudiant %d est:%s", i+1, Etudiant[i].nom); }
}
```

```
Invite de commandes - tc
Donner le nom de l etudiant 1 : Alaoui
Donner le nom de l etudiant 2 : Malloki

Le nom de l etudiant 1 est : Alaoui
Le nom de l etudiant 2 est : Malloki_
```

Définir une structure par la close typedef

- Pour ne pas être obligé de mettre à chaque fois dans la déclaration de la structure le mot clé **struct**, on utilise l'instruction **typedef** et un **type équivalent** dans la définition:

Exemple I-12 :

```
typedef struct EtudiantRepere{
   char nom[30], prenom[30];
   int age, note;
}TypeEquiv; //déclaration d'un type équivalent à struct EtudiantRepere
TypeEquiv Etudiant; //déclaration d'une variable de ce type
```

Exemple I-12 (suite):

```
void main(){
 printf("Donner le nom de l'étudiant : ");
 scanf("%s", Etudiant.nom);
 /*Affichage*/
 printf("\nle nom de l'étudiant est : %s", Etudiant.nom);
}
```


Structures imbriquées (méthode1)

- Imaginer qu'à l'intérieur de la structure **EtudiantRepere**, nous avons besoin de la date de naissance de l'étudiant. Cette date elle-même est une structure (**DateNaisRepere**) comportant trois champs correspondant au **jour**, **mois** et **annee**.

Exemple I-13 :

```
//Définition de la structure à utiliser (structure1)
typedef struct DateNaisRepere{
   int jour, mois, annee;
}DateNais; //déclaration d'un type équivalent à struct DateNaisRepere
```

Exemple I-13 (suite) :

```
//Définition de la structure principale (structure2)
typedef struct EtudiantRepere{
  char nom[30],prenom[30];
  int age,note;
   DateNais Date; //déclaration de 1 variable de type DateNais
}TypeEquiv; //déclaration d'un type équivalent à struct EtudiantRepere
TypeEquiv Etudiant; //déclaration de 1 variable de ce type
```

Structures imbriquées (méthode1)

- Voyez l'utilisation de la structure:

Exemple I-13 (suite) :

```
void main(){
  printf ("Donner le jour de naissance de l'étudiant : ");
  scanf("%d", &Etudiant.Date.jour);
  printf ("Donner le mois de naissance de l'étudiant : ");
  scanf("%d", &Etudiant.Date.mois);
  printf ("Donner l'année de naissance de l'étudiant : ");
  scanf("%d", &Etudiant.Date.annee);
  /*Affichage*/
  printf("\nle jour de naissance est:%d", Etudiant.Date.jour);
  printf("\nle mois de naissance est:%d", Etudiant.Date.mois);
  printf("\nl année de naissance est:%d", Etudiant.Date.annee);
}
```

Remarque

Dans la définition des structures, l'ordre est important !!

Structures imbriquées (méthode2: TD)

- Imaginer qu'à l'intérieur de la structure **EtudiantRepere**, nous avons besoin de la date de naissance de l'étudiant. Cette date elle-même est une structure (**DateNaisRepere**) comportant trois champs correspondant au **jour**, **mois** et **annee**.

Exemple I-14 :

```
//Définition de la structure1 à utiliser
struct DateNaisRepere{
   int jour, mois, annee;
};
```

Exemple I-14 (suite) :

```
//Définition de la structure2 (principale)
struct EtudiantRepere{
  char nom[30], prenom[30];
  int age, note;
  struct DateNaisRepere Date; //déclaration de 1 var de type DateNaisRepere
}Etudiant; //déclaration de 1 variable de type EtudiantRepere
struct EtudiantRepere Etudiant; //une autre façon de déclaration
```

Structures imbriquées (méthode2: TD)

- Voyez la fonction principale:

Exemple I-14 (suite):

```
void main(){
  printf ("Donner le jour de naissance de l'étudiant : ");
  scanf("%d", &Etudiant.Date.jour);
  printf ("Donner le mois de naissance de l'étudiant : ");
  scanf("%d", &Etudiant.Date.mois);
  printf ("Donner l'année de naissance de l'étudiant : ");
  scanf("%d", &Etudiant.Date.annee);
  /*Affichage*/
  printf("\nle jour de naissance est:%d", Etudiant.Date.jour);
  printf("\nle mois de naissance est:%d", Etudiant.Date.mois);
  printf("\nl année de naissance est:%d", Etudiant.Date.annee);
}
```

Remarque

Dans la définition des structures, l'ordre est important !!

En résumé

- Nous avons découvert qu'une **structure** permet de désigner sous un seul nom un ensemble de valeurs pouvant être de types différents. Par conséquent, de pallier le défaut dû à l'emploi des tableaux.

- Cette fois, l'accès à chaque élément de la structure se fait par son **nom** contrairement aux tableaux qui utilisent les indices.

Partie III:
La gestion de fichiers

Gestion de fichiers

- **Introduction**
- Le type FILE
- Manipulation des fichiers

Introduction

- Nous avons l'habitude d'échanger des informations entre le **programme** et l'**utilisateur** avec les **entrées-sorties conversationnelles**. Nous vous proposons ici d'étudier les fonctions permettant d'échanger des informations entre le **programme** et les **fichiers**.

- Le terme «**fichier**» désigne un ensemble d'informations situé sur une **mémoire de masse** (disque dur, disquette, CD…), sous forme du texte, des images, de la vidéo, …

- En C, tous les **périphériques**, qu'ils soient d'archivage (disque, disquette, ...) ou de communication (clavier, écran, imprimante...), peuvent être considérés comme des **fichiers.**

Type FILE

- Pour pouvoir travailler avec un **fichier**, le programme a besoin de quelques informations :
  - Adresse de la mémoire tampon du fichier;
  - Position actuelle de la tête de lecture/écriture du fichier;
  - Type d'accès au fichier; etc…
- Les informations du **fichier** sont rassemblées dans un **modèle de structure** de type spécial **FILE** défini dans <**stdio.h**>.
- Si on veut manipuler un fichier, il suffit de déclarer un **pointeur** (**pFile** par exemple) de type **FILE** et l'employer à la place du nom de fichier. Voyez la syntaxe de cette déclaration:

Syntaxe

```
FILE * pFile;
```

Remarque

•L'écriture de **FILE** est différente donc de l'écriture de **File** ou de **file** **!**

End Of File et la fonction feof()

- Lors de la fermeture d'un fichier ouvert en écriture, la **fin du fichier** est marquée automatiquement par le symbole (caractère invalide) **EOF** (End Of File).

Remarque

La valeur EOF est définie comme valant **−1** dans le fichier en-tête **stdio.h**.

- Lors de la lecture d'un fichier référencé par **pFile**, une fonction **feof**(**pFile**) teste l'indicateur de fin de ce fichier. Voyez son prototype:

Prototype

```
int feof(FILE *pFile);
```

- **Valeurs de retour:**

• **= 0 si la tête de lecture du fichier n'est pas arrivée à la fin du fichier (EOF)**
• **≠ 0 si la tête de lecture du fichier est arrivée à la fin du fichier.**

Remarque

Bien programmer, c'est toujours tester la valeur de retour des fonctions ! !

Les fonctions fopen() et fclose()

- L'ouverture et la fermeture d'un fichier s'effectue grâce à la fonction **fopen()** et **fclose()**. Voyez les prototypes suivants :

Prototype

```
FILE *fopen(char *nom, char *mode);
int fclose(FILE *pFile);
```

- Où **nom** est une variable pointeur vers une chaîne de caractères, le nom physique du fichier, et **mode** un pointeur vers une autre chaîne de caractères, qui définit le mode d'accès.
- **Arguments des fonctions:**
  - **nom**: le nom du fichier à ouvrir;
  - **mode**: le mode d'ouverture du fichier;
  - **pFile**: le pointeur qui représente le fichier.
- **Valeurs de retour:**

- **fopen(): ≠ NULL si fichier ouvert , = NULL en cas d'échec de l'ouverture.**
- **fclose(): = 0 si fichier fermé, ≠ 0 en cas d'échec de la fermeture.**

Les fonctions fopen() et fclose()

- Modes d'ouverture des fichiers sont:

  - "r" (Read): lecture seule dans un fichier préexistant.
  - "r+" : lecture / écriture dans un fichier préexistant.

  - "w" (Write): écriture seule dans un nouveau fichier. Si le fichier existe déjà son contenu est détruit.
  - "w+" : lecture / écriture dans un nouveau fichier. Si le fichier existe déjà son contenu est détruit.

  - "a" (Append): écriture à la fin du fichier seulement ou ajout.
  - "a+" : lecture / écriture à la fin du fichier seulement.

Remarque

Si la lecture ou l'écriture doit être fait en binaire, il faut ajouter le caractère "b" au mode d'ouverture des fichiers (exemple: "rb" ou "wb")

Les fonctions fopen() et fclose()

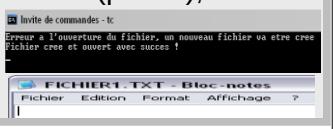
■ Voici un programme qui permet d'ouvrir le fichier **FICHIER1.TXT** en lecture. Si ce n'est pas possible, il va le créer et le fermer.

Exemple I-15

```
#include <stdio.h>…
void main(){
FILE *pFile1, *pFile2;
int c1,c2;
pFile1 = fopen("FICHIER1.TXT", "r");
if(pFile1 != NULL){
    printf("Fichier est bien ouvert !\n");
    c1= fclose(pFile1);
 }
else{
   printf ("Erreur a l'ouverture du fichier,
   celui-ci va être créer\n");
   pFile2 = fopen("Fichier1.txt","w+");
```

Exemple I-15 (suite)

```
 if(pFile2 == NULL){
    printf ("Impossible de créer
    le fichier !\n");
  }
  else{
     printf("Fichier créé et
     ouvert avec succès !\n");
   }
   c2= fclose(pFile2);
 }
}
```



Les fonctions fprintf() et fscanf()

■ Les fonctions **fprintf** et **fscanf** permettent d'effectuer respectivement **l'écriture** et **la lecture** dans un fichier dont le nom logique est **pFile**. Voyez les prototypes suivants :

Prototype

```
int fprintf(FILE *pFile, char *format, arguments);
int fscanf(FILE *pFile, char *format, arguments);
```

■ **Arguments des fonctions:**
- **pFile** : un pointeur qui représente le fichier;
- **format**: un pointeur vers une chaîne de format des données (%d, %c, %f, …);
- **arguments** : nombre d'argument qu'il faut (nom, prenom,…).

■ **Valeurs de retour:**
- **fprintf(): nombre de caractères écrits, valeur négative en cas d'erreur.**
- **fscanf(): nombre de caractères lus, valeur négative en cas d'erreur.**

Les fonctions fprintf () et fscanf()

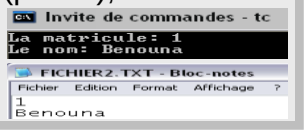
■ Voici un programme qui permet de saisir avec le clavier un enregistrement (matricule, nom) et l'écrire dans le fichier **FICHIER2.TXT**.

Exemple I-16

```
void main(){
 FILE *pFile2;
 int matricule;
 char nom[10];
 pFile2=fopen("FICHIER2.TXT","w");
 if(pFile2==NULL){
    printf("Erreur à l'ouverture
              du fichier");
 }
 else{
```

Exemple I-16 (suite)

```
/*Saisie de données à l'écran*/
 printf("La matricule:");
 scanf("%d", &matricule);
 printf("Le nom: ");
 scanf("%s", nom);
/*Saisie de données dans le fichier*/
 fprintf(pFile2,"%d\n%s\n",
          matricule, nom);
 fclose(pFile2);
 }
}
```



Les fonctions fprintf () et fscanf()

■ Voici un programme qui permet de lire les données du fichier **FICHIER2.TXT** et les afficher à l'écran:

Exemple I-17

```
void main(){
 FILE *pFile2;
 int matricule;
 char nom[10];
 pFile2=fopen("FICHIER2.TXT","r");
 if(pFile2==NULL){
    printf("Erreur à l'ouverture du
    fichier");
 }
 else{
```

Exemple I-17 (suite)

```
//Tant qu'on n'est pas à la fin du fichier
while(feof(pFile2)==0){
    /*On lit les enregistrements*/
    fscanf(pFile2,"%d\n%s\n",
            &matricule, nom);
   /*On affiche les enreg. à l'écran*/
    printf("%d\n%s\n",matricule,
           nom);
 }
 fclose(pFile2);
 }
}
```

Remarque
Les instructions `feof(pFile)==0;` et `! feof(pFile);` sont équivalentes

Les fonctions fputc() et fgetc()

■ La fonction **fputc()** permet d'écrire un seul caractère sur un fichier tandis que la fonction **fgetc()** permet de le lire. Voyez les prototypes suivants :

Prototype

```
int fputc (int caractere,  FILE *pFile);
int fgetc (FILE *pFile);
```

■ **Arguments des fonctions:**
- **caractere** : le caractère à écrire;
- **pFile** : le nom logique du fichier.

■ **Valeurs de retour:**

• fputc(): valeur ≥ 0 qui représente le caractère lu, valeur <0 en cas d'erreur.

• fgetc(): valeur ≥ 0 qui représente le caractère écrit, valeur <0 en cas d'erreur.

Les fonctions fputc() et fgetc()

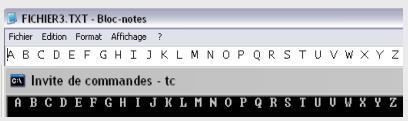
■ Voici un programme qui écrit tout l'alphabet (majuscule) dans le fichier **FICHIER3.TXT** caractère par caractère puis, affiche le contenu du fichier à l'écran caractère par caractère.

Exemple I-18

```
#include<stdio.h>…
void main(){
FILE *pFile;
char var1, var2; //variable type caractère
char var3=' '; //variable espace
pFile=fopen("FICHIER3.TXT","w");
if(pFile == NULL){
 printf("Erreur d'ouverture !");
}
else{
for(var1='A';var1<='Z';var1++){
    fputc(var1, pFile);
    fputc(var3, pFile);
 }
fclose(pFile);
}
```

Exemple I-18 (suite)

```
//Affichage du contenu du fichier
pFile=fopen("FICHIER3.TXT","r");
var2='0'; //initialisation
while(feof(pFile)==0){
  var2=fgetc(pFile); //Lecture d'1 c. du fichier
  printf("%c", var2); //Affichage d'1 c. à l'écran
}
fclose(pFile);
getch();
}
```



Les fonctions ftell() et fseek()

■ La fonction **ftell()** permet de connaître la position actuelle du curseur tandis que **fseek()** permet de se déplacer dans un fichier. Voyez les prototypes suivants :

Prototype

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

■ **Arguments des fonctions:**
- **déplacement**: le nombre de **pas** (octets) fait par le curseur (vers la fin du fichier s'il est <0 et vers le début du fichier s'il est <0).
- **origine**: position utilisée pour le déplacement.

■ L'argument **origine** peut prendre trois valeurs: 0,1, 2.
- ❑ origine = 0 ou SEEK_SET pour le début du fichier.
- ❑ origine = 1 ou SEEK_CUR pour la position courante.
- ❑ origine = 2 ou SEEK_END pour la fin du fichier.

■ **Valeurs de retour:**
- **fseek(): = 0 si elle a bien fonctionné, ≠ 0 en cas d'erreur.**
- **ftell(): = position en octets depuis le début du fichier, =-1 en cas d'erreur.**

Les fonctions ftell() et fseek()

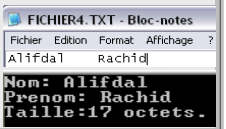
■ Le programme suivant écrit dans un nouveau fichier (**FICHIER4**) un **nom** puis, fait un déplacement de **4 pas** en avant et écrit un **prénom.** Par la suite, il détermine la **taille** du fichier en **octets**.

Exemple I-19

```
void main(){
FILE *pFile;
char nom[10], prenom[10];
long taille; //taille du fichier
pFile=fopen("FICHIER4.TXT","w");
if(pFile==NULL){
  printf("Erreur à l'ouverture!");
}
else{
//écrire le nom dans FICHIER4
printf("Nom: ");
scanf("%s", nom);
fprintf(pFile,"%s", nom);
```

Exemple I-19 (suite)

```
//écrire le prénom dans FICHIER4
printf("Prenom: ");
scanf("%s", prenom);
//effectuer 4 pas en avant
fseek(pFile, 4*sizeof(char), 2);
fprintf(pFile,"%s", prenom);
//déterminer la taille du FICHIER4
taille=ftell(pFile);
printf("Taille:%ld octets\n", taille);
fclose(pFile);
}
getch();
}
```



En résumé

- Nous avons vu que la **gestion de fichiers** est un cas général d'échange d'informations entre le **programme** et l'**utilisateur** qui englobe les **entrées-sorties conversationnelles**.

- Heureusement, en C on dispose des fonctions telles que **fopen**, **fclose**, **fscanf**, **fprintf**, **fseek**, etc, qui facilite ce type d'échange d'informations.