Table des matières

A.Principe du pointeur....4
1.Qu'est ce qu'un pointeur ? 4
a.Mémoire RAM 4
b.Une variable pointeur 5
c.Quatre opérateurs 5
d.Trois utilisations fondamentales des pointeurs....5
2.Avoir un pointeur dans un programme 6
a.Déclarer un pointeur....6
3.Fonctionnement des quatre opérateurs....7
a.Opérateur adresse : & 7
b.Opérateur étoile : *....7
c.Opérateur flèche : ->....8
d.opérateur crochet : [ ]....9
e.Priorité des quatre opérateurs 10
4.Allouer dynamiquement de la mémoire....10
a.La fonction malloc()....10
b.Libérer la mémoire allouée : la fonction free()....11
c.Le pointeur générique void* 12
d.La valeur NULL....12
5.Attention à la validité d'une adresse mémoire....13
a.Validité d'une adresse mémoire....13
b.Pourquoi caster le retour des fonctions d'allocation ?....14
6.Cas des tableaux de pointeurs....15
a.Une structure de données très utile....15
b.Un tableau de chaines de caractères....16
c.Utiliser les arguments de lignes de commandes....17
7.Expérimentation : base pointeur....18
8.Mise en pratique : base pointeurs....21
a.Avoir des pointeurs et les manipuler 21
b.Tests tableaux / pointeurs....21
c.Base allocation dynamique....23
d.Attention aux erreurs....24

e.Tableaux de chaines....24
B.Allocation dynamique de tableaux....25
1.Allouer un tableau avec un pointeur....25
2.Allouer une matrice avec un pointeur de pointeur....25
3.Différences entre tableaux statiques et dynamiques....28
4.Autres fonctions d'allocation dynamique....28
a.Fonction calloc() 28
b.Fonction realloc() 29
5.Mise en pratique : Allocation dynamique 29
a.Allouer dynamiquement des tableaux....29
b.Allouer dynamiquement des matrices....31
c.Allocation dynamique calloc() et realloc()....32
C.Pointeurs en paramètre de fonction 33
1.Passage par référence 33
a.Cas général d'une variable quelconque 33
b.Exemple pour avoir l'heure....34
c.Passage par référence d'une structure 35
d.Passage par référence d'une variable pointeur....36
2.Tableaux dynamiques en paramètre....38
3.Mise en pratique : Passage par référence....40
a.Passage par référence, base....40
b.Passage par référence, opérateurs bit à bit....41
c.Passage de pointeurs par référence....42
d.Passage de tableaux dynamiques 43
D.Fichiers (type FILE*)....44
1.Base fichier....44
a.Le type FILE*....44
b.Ouverture et fermeture d'un fichier....44
c.Spécifier un chemin d'accès....45
2.Fichiers binaires 46
a.Écriture et lecture en mode binaire 46
b.Détecter la fin d'un fichier binaire....47
c.Déplacements dans un fichier....48
3.Écriture et lecture en mode texte....48

Chapitre 4 : Variables pointeurs

a.Détecter la fin d'un fichier, EOF et feof()..49
b.Lecture / écriture de caractères...49
c.Lecture / écriture de chaînes...50
d.Lecture / écriture formatées...51
4.Sauvegarde d'éléments dynamiques...52
a.Sauver et récupérer un tableau dynamique...52
b.Récupérer des données via des pointeurs...53
5.Mise en pratique : Fichiers...54

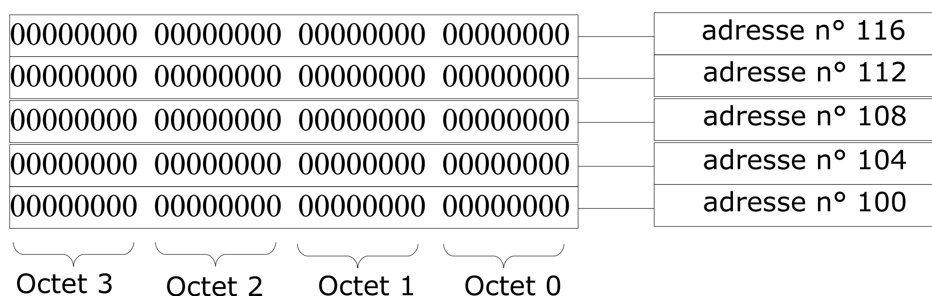
A. Principe du pointeur

1. Qu'est ce qu'un pointeur ?

a. Mémoire RAM

Tous les objets informatiques que nous avons évoqués, qu'il s'agisse des variables, des fonctions, des tableaux, des structures etc., tous correspondent à une inscription localisée quelque part dans la mémoire principale, dénommée également mémoire vive ou RAM. En un mot chaque objet a une "adresse" dans la mémoire principale, cette adresse est un nombre et il correspond à un emplacement de la mémoire.

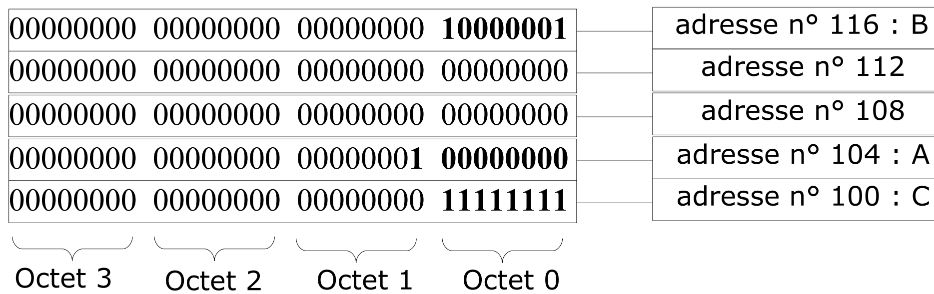
La mémoire en 32 bits est construite sur la base d'un entrelacement de 4 octets en 4 octets consécutifs. A partir de l'octet qui est la plus petite unité de mémoire adressable (l'octet a 8 bits et correspond en C au type char), la mémoire est constituée d'emplacements dits "mots" de quatre octets, 32 bits, dont l'adresse est toujours un nombre multiple de quatre généralement en hexadécimal. Voici une représentation de ce principe en partant par exemple de l'adresse "100" avec une progression de 4 en 4 ici en décimal pour simplifier :



Dans un programme lorsqu'une variable quelconque est déclarée une adresse mémoire lui est automatiquement affectée à la compilation. Soit par exemple trois variables dans un programme :

```
char C=255;
int A=256, B=129;
```

Admettons que l'adresse de C soit 100, l'adresse de A pourrait être 104 et l'adresse de B 116 ce qui donnerait en mémoire :



la variable B a pour valeur 129 et est à l'adresse 116

la variable A a pour valeur 256 et est à l'adresse 104

la variable C a pour valeur 255 et est à l'adresse 100

Chaque nouvel objet commence toujours sur une frontière de mot. Seul des objets de type char (1 octet) ou éventuellement short (2 octets) peuvent posséder des adresses intermédiaires d'octets. C'est pourquoi l'adresse de la variable entière A est sur la frontière de mot qui suit l'adresse de la variable de type char C. Les trois octets d'adresses 101, 102 et 103 ne sont pas utilisés.

b. Une variable pointeur

Un pointeur est *une variable qui prend pour valeur des adresses mémoire*. De ce fait une variable pointeur permet d'accéder au contenu de la mémoire à partir de l'adresse qu'elle contient. En elle-même la variable pointeur est codée sur quatre octets et possède, comme toute variable, sa propre adresse mémoire.

c. Quatre opérateurs

Pour utiliser des adresses mémoire il y a deux opérations essentielles :

- □ récupérer une adresse mémoire
- □ accéder à une adresse mémoire

L'opérateur de référence : &, "adresse de" permet de récupérer une adresse mémoire (c'est celui qui est utilisé avec la fonction scanf())

L'opérateur d'indirection : * , "étoile" permet d'accéder à une adresse mémoire. Il y a deux autres opérateurs pour accéder à des adresses mémoire qui sont en fait des contraction d'écriture : l'opérateur → "flèche" qui permet d'accéder aux champs d'une structure à partir de son adresse mémoire et l'opérateur [ ] crochet qui permet d'accéder aux éléments d'un tableau à partir de l'adresse du premier élément.

d. Trois utilisations fondamentales des pointeurs

Les pointeurs constituent l'un des plus puissants outils du langage C. Il y a trois cas d'utilisation de pointeurs :

Allocation dynamique de tableaux

Il est possible avec un pointeur de réserver un espace mémoire contiguë de n'importe quelle taille pendant le fonctionnement du programme et d'utiliser ce bloc comme un tableau. La première utilisation de pointeurs est ainsi de pouvoir obtenir dynamiquement des tableaux de n'importe quelle taille et de n'importe quel type. (Détails dans le chapitre Allocation dynamique de tableaux)

pointeurs en paramètre de fonction

Avec un pointeur il est possible d'accéder à une variable via son adresse mémoire. De ce fait, avec un pointeur en paramètre de fonction, il est possible de communiquer à une fonction l'adresse mémoire d'une variable et dans la fonction d'accéder à cette variable et de modifier sa valeur via son adresse mémoire. C'est à dire qu'avec un pointeur il est possible de transformer les paramètres d'entrées en sortie. (Détails dans le chapitre pointeurs en paramètre de fonction).

Structures de données composées (listes chainées arbres, graphes)

Avec des pointeurs il est possible d'élaborer des structures de données complexes qui n'existent pas en tant que telle dans le langage : les listes chaînées, les arbres et les graphes. Ces grandes figures de la programmation et de l'algorithmique sont constructibles dynamiquement en C via l'utilisation de pointeurs. (Un chapitre est consacré aux listes)

2. Avoir un pointeur dans un programme

a. Déclarer un pointeur

Pour déclarer un pointeur il faut :

- le type de l'objet sur lequel pointer
- l'opérateur * (à gauche du nom du pointeur)
- un nom (identificateur) pour le pointeur

Par exemple :

```
char *c;            // déclare un pointeur sur char
int *i, *j;         // déclare deux pointeurs sur int
float *f1,*f2;      // déclare deux pointeurs sur float
```

c est un pointeur de type char* : il peut contenir des adresses de char.

i et j sont deux pointeurs de type int*, ils peuvent contenir des adresses de int

f1 et f2 sont deux pointeurs de type float*, ils peuvent contenir des adresses de float

Avec des structures c'est identique :

```
typedef struct player{  // définition du type player
    int x,y;
    int dx,dy;
}player;

player *p;              // déclare un pointeur player
```

p est un pointeur de type player* et p peut contenir des adresses de structures player.

3. Fonctionnement des quatre opérateurs

a. Opérateur adresse : &

L'opérateur & accolé à gauche d'un objet quelconque dans un programme retourne son adresse. Par exemple :

```
int k;
   &k      // cette expression vaut l'adresse mémoire de la
           // variable de k
```

Toute adresse est une valeur pointeur : si *k* est un objet de type *T* alors &*k* est de type pointeur sur *T*.

Remarque :

Notons que *k* doit être ce que l’on appelle une lvalue. C’est à dire une expression qui possède une adresse mémoire accessible, en général une expression à laquelle il est possible d’affecter une valeur. En effet, soit i un int, &i est l’adresse de i qui pointe sur i. Mais une expression comme &(i+1) avec les parenthèses n’est pas correcte syntaxiquement car (i+1) ne correspond pas à une variable en mémoire, ce n’est pas une lvalue. Il n'est pas possible non plus d'écrire quelque chose comme : (i+1)=77;

Le programme suivant permet d’afficher les adresses mémoire de plusieurs variables en utilisant l’opérateur & (et le format %p de printf() réservé pour afficher des valeurs de pointeur c'est à dire des adresses mémoire) :

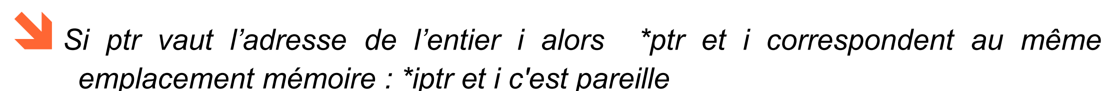
```
int main()
{
char c;
int i,j;
struct {
    int x,y;
    float dx,dy;
}s1;
   printf("l’adresse de c est : %p\n", &c);
   printf("l’adresse de i est : %p\n", &i);
   printf("l’adresse de j est : %p\n", &j);
   printf("l’adresse de s1 est : %p\n", &s1);
   return 0;
}
```

Les expressions, &c, &i, &j, &s1 correspondent respectivement aux adresses des objets c, i, j, s1, ce programme permet de les visualiser.

b. Opérateur étoile : *

L'opérateur * accolé à gauche d'un pointeur qui contient l’adresse mémoire d’un objet, retourne l’objet qui est à cette adresse. Cet opérateur permet d'accéder à l'objet via son adresse et de le modifier. Par exemple :

```
int i;
int* ptr = &i;
```



*Si ptr vaut l’adresse de l’entier i alors *ptr et i correspondent au même emplacement mémoire : *iptr et i c'est pareille*

Le programme suivant permet de visualiser l'adéquation qu'il y a, du fait de l'utilisation de l'opérateur *, entre un objet quelconque et un pointeur qui contient son adresse :

```
int main()
{
int i = 23;                    // 1
int  *ptr;

   ptr = &i;                   // 2
   printf("%d, ",*ptr);

   *ptr = 55;                  // 2
   printf("%d, ",i);

   i = 777;                    // 3
   printf("%d.",*ptr);
   return 0;
}
```

Le programme imprime : 23, 55, 777.

(1) Déclarations de l'entier i qui vaut 23 et du pointeur d'entier ptr

(2) l'adresse de i, c'est-à-dire l'expression &i, est affectée au pointeur ptr.

A partir du moment où ptr contient l'adresse de la variable i, *ptr et i correspondent au même emplacement mémoire, i et *ptr sont identiques, c'est le même objet. Donc afficher *ptr c'est afficher i, la valeur 23.

(3) La valeur 55 est affectée à *ptr, c'est à dire à i et i vaut maintenant 55.

(4) La valeur 777 est affectée à la variable i et la valeur de l'expression *ptr est également 777

c. Opérateur flèche : ->

L'opérateur flèche, ->, concerne uniquement les structures. La flèche est simplement une facilité d'écriture, c'est la contraction des deux opérateurs étoile et point utilisés ensemble.

Soit la définition d'un type de structure nommé "ennemi" :

```
typedef struct{
    int vie;
    float x,y,dx,dy;
    int color;
}ennemi;
```

Deux variables du type ennemi:

```
ennemi S1,S2;
```

Une variable pointeur de type ennemi* :

```
ennemi*p;
```

avec

```
p=&S1;
```

Si l'on utilise l'opérateur étoile pour accéder aux champs de la structure S1 à partir du pointeur p, on a :

```
(*p).vie=1;
(*p).x=rand()%80;
(*p).y=rand()%25;
 etc.
```

Comme vu précédemment le pointeur p contient l'adresse de S1 et l'expression *p est équivalente à S1. *p peut s'utiliser comme S1 avec l'opérateur point pour accéder aux champs de la structure. Mais attention il faut lui ajouter des parenthèses parce que l'opérateur point est prioritaire par rapport à l'opérateur étoile. En effet l'expression *p.vie est équivalente à *(p.vie) elle signifierait d'une part que p est une structure et non un pointeur et d'autre part que le champ vie est un pointeur, ce qui n'est pas le cas.

Afin d'éviter des erreurs et de faciliter l'utilisation des pointeurs de structure l'expression p-> est une notation abrégée de (*p). et pour accéder aux champs de la structure ennemi on écrit simplement :

```
p->vie=1;
p->x=rand()%80;
p->y=rand()%25;
etc.
```

d. opérateur crochet : [ ]

Un tableau est l'adresse mémoire du premier élément du tableau :

```
int tab[50];     //  tab vaut &tab[0]
```

Puisqu'un tableau est une adresse mémoire il est possible de l'affecter à un pointeur

```
int *p;
    p=tab;
```

ce qui signifie ensuite que tab et p sont équivalents et il est possible d'utiliser p à la place de tab :

```
int i;
    for (i=0; i<50; i++)
       p[i] = rand()%256;
```

et nous pouvons écrire également :

```
for (i=0; i<50; i++)
    *(p+i) = rand()%256;
```

L'expression (p+i) vaut l'adresse mémoire de p, début du tableau, plus un déplacement de i*sizeof(int) octets en mémoire, soit l'adresse des éléments suivants. Pour i=0 c'est l'adresse du tableau, l'élément 0, pour i=1 c'est l'adresse de l'élément suivant quatre octets plus loin, pour i=2 c'est l'adresse de l'élément 8 octets plus loin etc. Ainsi *(p+i) permet d'accéder à chaque éléments i du tableau.

Mais cette écriture est peu usitée, en général ce sont les opérateurs crochets qui sont utilisés p[i] est une contraction de *(p+i)

Remarque :

Comme nous le voyons il est possible d'utiliser les opérateurs arithmétiques avec les pointeurs mais attention il est strictement interdit d'accéder à des adresses mémoire non réservées sous peine de plantage ou de comportements incertains du programme.

e. Priorité des quatre opérateurs

La flèche et le point pour l'accès aux champs d'une structure, l'opérateur crochet pour l'indiçage de tableau et les parenthèses pour l'appel de fonction sont du plus fort niveau de priorité. L'adresse, qui donne la référence mémoire d'un objet et l'étoile qui permet d'accéder à un objet via son adresse sont de niveau juste inférieur (annexe 1 : priorité des opérateurs)

4. Allouer dynamiquement de la mémoire

a. La fonction malloc()

Nous avons vu des cas où un pointeur reçoit comme valeur l'adresse d'une variable préalablement déclarée. Mais le pointeur permet également d'obtenir dynamiquement, pendant le fonctionnement du programme, une adresse mémoire allouée pour une variable de n'importe quel type. Pour ce faire le langage C fournit une fonction importante :

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

Cette fonction alloue un bloc de mémoire de *taille* octets passée en argument et retourne l'adresse mémoire de la zone allouée (ou NULL si la mémoire est insuffisante). Le void* est un pointeur générique, c'est à dire un pointeur qui peut fonctionner avec n'importe quel type d'objet. Le fonctionnement du système garantit que l'adresse allouée est bien réservée pour l'objet demandé par le programme et qu'elle ne pourra en aucun cas servir à autre chose. La taille d'une zone mémoire pour un objets quelconque de type *T* s'obtient tout simplement avec l'opérateur sizeof (*T*). Ainsi, Soit ptr un pointeur de type *T**, l'appel de la fonction est :

```
ptr = malloc (sizeof( T )) ;
```

Et on a dans ptr l'adresse d'une zone mémoire de taille suffisante réservée pour un objet de type *T*. Par exemple :

```
int main()
{
int*iptr;
float* tab[10];        // tableau de pointeurs float*
int i;

   iptr=malloc(sizeof(int));
   *iptr=45;
   printf("*iptr, a l'adresse %p  vaut : %d\n", iptr, *iptr);

   srand(time(NULL));
   for(i=0; i<10; i++){
      tab[i]=malloc(sizeof(float));
          *tab[i]= (rand() / (float)RAND_MAX)*5;
          printf("*tab[%d] a l'adresse %p vaut : %.2f\n",
                 i,tab[i], *tab[i]);
   }
   return 0;
}
```

Dans ce programme il y a un pointeur d'entier et un tableau de pointeurs de float. Tout d'abord une adresse mémoire est allouée pour la variable pointeur d'entier iptr

et la valeur 45 est affectée à cette adresse dans la mémoire. L'adresse obtenue pour iptr et la valeur affectée à cette adresse sont affichées. Ensuite une adresse mémoire est allouée à chacun des pointeurs du tableau de pointeurs. A chacun des emplacements mémoire correspondants une valeur aléatoire flottante entre 0 et 5 est affectée. L'adresse affectée à chaque pointeur est affichée ainsi que la valeur aléatoire obtenue.

b. Libérer la mémoire allouée : la fonction free()

La mémoire allouée est réservée, c'est-à-dire inutilisable pour autre chose dans le programme, c'est pourquoi il est important de la libérer lorsque l'objet alloué n'est plus utilisé dans le programme afin de rendre cette mémoire à nouveau disponible. Cette opération inverse de l'allocation s'effectue avec un appel à la fonction

```
void free(void*p);
```

Par exemple pour libérer la mémoire allouée au pointeur ci-dessus l'appel ci-dessous est nécessaire :

```
free(ptr);
```

En C, pendant le fonctionnement d'un programme, il n'y a pas de ramasse-miettes qui fasse le travail automatiquement comme en java ou en C# par exemple. Le risque est tout simplement d'épuiser la mémoire RAM disponible et que le programme manque de mémoire pour pouvoir fonctionner à un moment donné. C'est possible en particulier avec de petits objets numériques en robotique ou ailleurs. Par exemple dans le programme ci-dessous :

```
int main()
{
double*p[500000];
int i;
int fin=0;
   do{
        printf("\n\n");

        for (i=0; i<500000; i++)
             p[i]=malloc(sizeof(double)) ;        // 1

        for (i=0; i<500000; i++)                  // 2
             free(p[i]) ;

        printf("Play again ? (o/n)\n");
            fin=getch();

   }while(fin!='n');
   return 0;
}
```

A chaque tour de boucle 500000 adresses mémoire sont allouées (1) pour un total de 4 millions d'octets (500000*8octets). Si la mémoire n'est pas libérée, chaque tour ajoute encore et au bout d'un moment le programme se bloque. En revanche si la mémoire allouée est libérée (2) le programme peut tourner indéfiniment.

Disons qu'il y a aussi une question de savoir vivre du programmeur C qui, lorsque le programme quitte, doit faire attention à ce que son programme laisse la mémoire dans l'état où il l'a trouvée au lancement. A la sortie du programme, par principe, il faut désallouer toute la mémoire précédement allouée.

c. Le pointeur générique void*

La fonction malloc() ci-dessus renvoie un pointeur de type void*.

Le pointeur void* est un un pointeur de type indéfini, son utilisation principale est d'en faire un pointeur dit "générique" à savoir un pointeur indépendant de tout type en particulier, juste une adresse mémoire pour un espace mémoire contigu d'une taille donnée. Un tel pointeur permet de manier l'adresse mémoire d'un objet dont on ignore le type.

C'est bien le cas avec la fonction malloc() qui est utilisée pour n'importe quel type d'objet, int*, char*, float*, double* ainsi que pour des adresses mémoire de structures créées par le programmeur. La fonction doit pouvoir allouer de la mémoire pour n'importe quel type d'objet. C'est pourquoi la fonction malloc() renvoie un pointeur générique, c'est-à-dire l'adresse mémoire d'une zone de mémoire contiguë de la taille en octets passée en argument à la fonction.

void inutilisable avec l'opérateur * sans cast dans un programme*

Par ailleurs, toutes les opérations avec l'opérateur * étoile sur un pointeur générique void* nécessiterons un cast pour pouvoir être effectuées, par exemple :

```
int toto=55;
void* ptr = &toto;
    printf("toto=%d\n",*((int*)ptr)); // ça marche avec le cast
    printf( "toto=%d\n", *ptr);        // sinon erreur à la
                                       // compilation.
```

d. La valeur NULL

NULL est défini dans <stddef.h>. C'est une valeur spéciale de pointeur qui vaut en fait 0. Le C garantit que 0 ne sera jamais une adresse utilisable par un objet quelconque. Et 0, qui n'est donc pas une adresse mémoire, est le seul cas où un entier peut être affecté à un pointeur. Il s'agit en fait d'une conversion de type de la forme ((void*) 0).

De ce fait il est fréquent de rencontrer des tests de comparaison d'un pointeur avec la valeur NULL. Par exemple, si la fonction malloc() ne peut pas allouer suffisamment de mémoire elle renvoie NULL et il est possible de garantir que le programme fonctionnera même si la mémoire est insuffisante avec un test sur la valeur de retour :

```
ptr=(int*)malloc(sizeof(int));
if (ptr!=NULL){
   printf("malloc réussi, le pointeur peut être utilisé\n");
   *ptr=rand()%300;
 }
 else
   printf("pas assez de mémoire,"
          "le pointeur ne peut pas être utilisé\n");
```

D'autre part, il est très souvent nécessaire d'initialiser ses pointeurs à NULL lors de leur déclaration dans un programme afin d'éviter d'essayer d'accéder par mégarde à des zones mémoire non réservée (les valeurs résiduelles contenues par un pointeur non initialisé), ce qui plante le programme.

5. Attention à la validité d'une adresse mémoire

a. Validité d'une adresse mémoire

L'allocation dynamique nécessite de la part du programmeur beaucoup d'attention. C'est lui qui a à charge l'allocation et la libération de la mémoire.

En effet lorsqu'un tableau statique ou n'importe quelle variable est déclaré dans un programme, l'espace mémoire requis est automatiquement réservé par le compilateur au moment de la compilation. C'est pourquoi il est possible d'écrire dans cet espace et d'utiliser le tableau ou la variable. En revanche ce n'est pas le cas avec un pointeur.

Lorsqu'un pointeur est déclaré dans le programme il ne contient pas l'adresse mémoire d'un bloc réservé, il contient n'importe quoi, une adresse aléatoire, ce qui traîne dans la mémoire et si l'on tente d'écrire à une adresse mémoire dans un bloc non réservé le programme plante ou quitte brutalement avec un message d'erreur. Par exemple :

```
int main()
{
char*ptr;
char tab[80];
   printf("entrer une phrase :\n");
   fgets(tab,80,stdin);
   printf("tab : %s\n",tab);

   printf("entrer une autre phrase :\n");
   fgets(ptr,80,stdin);       // ERREUR !! le programme quitte
   printf("ptr : %s\n",ptr);
     return 0;
}
```

Le pointeur ptr ne contient pas d'adresse valide, il n'a pas fait l'objet d'une allocation, l'adresse qu'il contient n'est pas accessible en écriture. Tenter d'y accéder provoque une erreur à l'exécution. La difficulté c'est que ça compile très bien. Ce genre d'erreur n'est pas détecter par le compilateur.

En revanche si par exemple nous affectons à ptr l'adresse du tableau tab, ça marche :

```
int main()
{
char*ptr;
char tab[80];
   printf("entrer une phrase :\n");
   fgets(tab,80,stdin);
   printf("tab : %s\n",tab);

   ptr=tab;
   printf("entrer une autre phrase :\n");
   fgets(ptr,80,stdin);                     // ok
   printf("ptr : %s",ptr);                  //
   printf("tab : %s\n",tab);         // ptr et tab sont identiques
      return 0;
}
```

ptr prend l'adresse de tab qui est dûment réservée par la machine pour le tableau tab. Ptr et tab désignent alors le même espace mémoire, écrire à partir de ptr revient à écrire à partir de tab, c'est à la même adresse en mémoire, le même endroit dans la mémoire. Attention donc parce que dans ce cas ptr et tab sont deux accès pour le même bloc de mémoire et non deux blocs différents.

Une autre solution consiste à allouer dynamiquement un second bloc mémoire et y copier le contenu du bloc tab (voir section sur l'allocation dynamique de tableaux) :

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


int main()
{
char*ptr;
char tab[80];
int i;
   printf("entrer une phrase :\n");
   fgets(tab,80,stdin);

   // un nouveau bloc alloué
   ptr=(char*)malloc(sizeof(char)*strlen(tab)+1);

   // copie de tab
   strcpy(ptr,tab);

   // modification de la copie
   for(i=0; i<strlen(ptr); i++)
      ptr[i]++;

    // ptr et tab sont différents
   printf("ptr : %s\n",ptr);
   printf("tab : %s\n",tab);
    return 0;
}
```

Cette fois ptr et tab sont deux blocs distincts correspondant chacun à une adresse mémoire valide.

b. Pourquoi caster le retour des fonctions d'allocation ?

Le retour de la fonction malloc() par exemple peut être affecté à n'importe quel type de pointeur sans aucun contrôle de rien. Par exemple :

```
char c;
double *d;

    d=&c;
```

provoque un avertissement "warming" à la compilation : affectation d'un pointeur incompatible. &c est de type char* et d est de type double*. Il vaut mieux y faire attention parce que le pointeur d prend l'adresse d'un char c'est à dire d'un espace mémoire d'un seul octet réservé. Si nous écrivons en dehors de l'espace réservé le programme plante :

```
*d=512;          // fait planter le programme.
```

En revanche si nous écrivons :

```
double *d;
    d=malloc(sizeof(char));    // attention erreur invisible
```

La situation est identique. l'espace mémoire réservé fait un seul octet, alors qu'un double attend 8 octets mais il n'y a aucun avertissement ni message d'erreur de la part du compilateur ; ça passe discrètement et ça peut devenir un bogue infernal à trouver qui fait planter le programme de temps en temps.

Ainsi pour bénéficier d'un message d'avertissement sur la compatibilité du retour void* de la fonction malloc() avec le pointeur de destination, il est préférable de toujours caster le void* dans le type voulu. Ce cast s'opère en plaçant entre parenthèse le type de pointeur voulu à la gauche du retour de la fonction. Ainsi, si dans notre exemple nous écrivons :

```
double *d;
   // retour de malloc casté en char* :
   d=(char*)malloc(sizeof(char));   // alors l'erreur est rendue
                                    // visible à la compilation
```

Il y a un message d'avertissement "warming" du fait de l'incompatibilité entre le pointeur d double* et l'allocation pour un char*. D'autre part cette écriture oblige à bien être attentif à l'appel de malloc() effectué : la taille demandée doit être en accord avec le type retourné.

6. Cas des tableaux de pointeurs

a. Une structure de données très utile

Le tableau de pointeurs permet de stocker non plus des objets mais les adresses mémoires des objets. Par exemple soit une structure troll :

```
typedef struct troll{
    int x,y;
    int color;

}t_troll;
```

Nous pouvons définir un tableau statique de pointeurs, c'est un tableau ordinaire qui contient des pointeurs, en l'occurence des t_troll*

```
#define NBMAX   100

t_troll* tab[NBMAX];  // un tableau de pointeurs
```

Pour initialiser un tableau de pointeurs il faut veiller à ce que chaque pointeur contienne une adresse mémoire valide :

```
for (i=0; i<NBMAX; i++){

    tab[i]=(t_troll*)malloc(sizeof(t_troll)); // allocation mémoire

    // chaque élément ici est un pointeur de structure :
    tab[i]->x=rand()%800;
    tab[i]->y=rand()%600;
    tab[i]->color=rand()%256;
}
```

Par ailleurs c'est un tableau statique et il se comporte comme les autres en paramètre de fonction :

```
void initialisation(t_troll* t[])
```

```
{
int i;
    for (i=0; i<NBMAX; i++){

        t[i]=(t_troll*)malloc(sizeof(t_troll));
        t[i]->x=rand()%800;
        t[i]->y=rand()%600;
        t[i]->color=rand()%256;
    }
}
void affiche(t_troll*t[])
{
int i;
    for (i =0; i<NBMAX; i++){
       printf("%4d %4d %4d\n",t[i]->x,t[i]->y,t[i]->color);
    }
}
int main()
{
t_troll* ALL[NBMAX];

    initialisation(ALL);
    affiche(ALL);
   return 0;
}
```

Utiliser les adresses mémoire des objets plutôt que les objets en dur est avantageux surtout pour les paramètres des fonctions associées aux traitements de ces objets. Le fait de pouvoir passer une adresse mémoire en paramètre de fonction transforme l'entrée en sortie : il est possible d'écrire à l'adresse passée (voir la section pointeurs en paramètre de fonction, passage par référence).

b. Un tableau de chaines de caractères

Les chaines de caractères sont des tableaux de caractères et la chaine se termine avec le caractère '\0'. Les pointeurs char* sont souvent utilisés et ils contiennent alors l'adresse du début du tableau.

L'intérêt est alors de pouvoir constituer des ensembles de chaînes de caractères plus facilement. Une liste de mots devient un tableau de pointeurs de caractères. Par exemple :

```
char* liste_mots[]={"titi",
                    "totofait du vélo",
                    "tutu",
                    "tata go to the sea",
                    "fin" };
```

Avec une matrice de char aurions :

```
char mat_mots[][100]={ "titi",
                       "totofait du vélo",
                       "tutu",
                       "tata go to the sea",
                       "fin" };
```

Ce n'est pas la même chose en mémoire. La matrice de char est moins souple. Dans le cas du tableau de pointeurs de caractères la taille mémoire est exactement celle du nombre des caractères. Dans celui de la matrice de caractère, qui est un

tableau de tableaux de caractères, on a un bloc mémoire de 5*100 octets réservé quelque soit la taille des mots et il ne faut pas qu'un mot ou la phrase dépasse 100 caractères.

Lorsque la taille du tableau de char* n'est pas explicite, par exemple en cas de liste très longue de mots ou de phrases, le problème est de connaître la fin du tableau. Pour liste_mots et mat_mots le mot "fin" indique qu'il s'agit du dernier mot dans le tableau et les deux peuvent être parcourus de la façon suivante :

```
int i;
    for (i=0;  strcmp("fin", liste_mots[i]) != 0 ; i++)
        printf("%s\n",liste_mots[i]);
```

La fonction strcmp() compare deux chaînes de caractères et renvoie une valeur négative, nulle ou positive selon que la première est inférieure, égale ou postérieure dans l'ordre lexicographique.

Dans notre exemple la boucle va continuer jusqu'à arriver à l'indice de la chaîne "fin" (l'indice i est égal à 4) là la fonction strcmp() renvoie 0, le test devient faux et la boucle s'arrête.

Une autre méthode consiste à utiliser l'adresse 0 de la mémoire, la valeur NULL. La formule suivante permet d'obtenir une sentinelle à NULL dans un tableau de char* :

```
((char*)0)
```

La valeur 0, de type int, castée en char* est accepté dans le tableau de char* comme une chaine de caractères. La liste de mots peut s'écrire de la façon suivante :

```
char* liste_mots[ ] ={ "titi",
                        "toto fait du vélo",
                        "tutu",
                        "tata go to the sea",
                        ((char*)0) };
```

et le test de la boucle devient :

```
int i;
    for (i=0;  liste_mots[i] != NULL ; i++)
        printf("%s\n",liste_mots[ i ]);
```

c. Utiliser les arguments de lignes de commandes

Il est possible de communiquer des arguments aux paramètres du main() au moment du lancement de l'application via une fenêtre console (invite de commande). Les paramètres du main() lorsqu'ils sont présents sont :

```
int main(int argc, char* argv[])
{
    (...)
    return 0 ;
}
```

Toutes les informations sont transmises via le tableau de chaine de caractères argv et l'entier argc donne le nombre de chaines transmises. Il y a toujours au moins une chaine à l'indice 0, c'est le nom du programme et argc vaut toujours au moins 1.

A partir de là nous construire des application qui reçoivent au démarrage des informations. Par exemple :

#include <stdio.h>

```
#include <stdlib.h>


int main(int argc, char*argv[])
{
int i;
    printf("nom du programme : %s\n",argv[0]);
    if (argc>1)
        for (i=1;i<argc; i++ )
            printf("param %d : %s\n",i,argv[i]);

    // suite du programme qui prend ou non en compte
    // les arguments reçus.
   return 0;
}
```

Ce programme affiche son nom puis, s'il y a des arguments passés les affiche.

Sous windows l'invite de commandes (une fenêtre console) est accessible à partir du menu Démarrer dans le dossier "Accessoires". Pour lancer l'exécutable à partir de l'invite il faut accéder à l'exécutable en tapant tout son chemin exacte depuis la racine du disque. La commande cd permet de se déplacer dans l'arborescence, cd .. (deux points) pour remonter d'un niveau et cd /dossier1 pour descendre dans dossier1. Le mieux est pour commencer de copier son exécutable à la racine du disque C ensuite de remonter jusque à la racine (cd .. jusqu'à la racine)et d'appeler son programme (ici mon programme s'appelle test.exe) :

```
C:\> test.exe
```

Si vous tapez enter et que le programme est à la bonne place le programme est lancé. Comme il n'y a pas d'argument il se contente d'afficher son nom.

les arguments sont alors à placer après l'appel du programme, par exemple :

```
C:\> test.exe tata fait du velo
```

affiche :

```
nom du programme : test.exe
param1 : tata
param2 : fait
param3 : du
param4 : velo
```

il y a un découpage effectué avec les espaces. Pour l'éviter il suffit de regrouper les ensembles de mots par des guillemets :

```
C:\> test.exe "tata fait" "du velo"
```

affiche :

```
nom du programme : test.exe
param1 : tata fait
param2 : du velo
```

Bien entendu ces chaines peuvent contenir des nombres et être converties ensuite dans le programme (utiliser les fonctions standards adéquates atoi(), atof() etc.).

L'argument peut aussi être un nom de fichier qui contient beaucoup de chose pour le fonctionnement du programme etc..

7. Expérimentation : base pointeur

```
/**************************************************************

Qu'est ce qu'un pointeur ?
une variable qui prend uniquement des adresses mémoire pour
valeur.

A quoi servent les pointeurs ?
Trois utilisations :
1) en paramètre de fonction : permet de passer l'adresse mémoire
d'une variable et de transformer l'entrée en sortie (possibilité
d'écrire à une adresse mémoire)

2) allocation dynamique d'objets ou de tableaux d'objet

3) créer des structures de données dynamique non natives en C :
listes chainées, arbres, graphes

Comment les utilise t-on ?
Il y a quatre opérateurs associés et trois fonctions
d'allocation de mémoire (pour l'allocation dynamique)

Les quatre opérateurs sont :
1) "adresse de" : & permet de récupérer l'adresse mémoire d'une
variable

2) "étoile" : * permet de déclarer un pointeur etd'accéder à une
adresse mémoire.

EXEMPLE :
int a;
int *ptr; // déclaration d'un pointeur d'entier
    ptr = &a; // ptr prend pour valeur l'adresse de a
*/

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

int main()
{
int a;
int*ptr=&a;

   a= 100;
   *ptr=200;
   printf("a : %d / *ptr : %d\n",a,*ptr);

   return 0;
}

/**************************************************************

3) "flèche" : -> permet daccéder à un champ de structure via un
pointeur

EXEMPLE :
struct test{
   int x,y;
} t;

struct test *ptr;
```

```
   ptr=&t;
   ptr->x=450;
   ptr->y=90;

équivaut à : (*ptr).x=450 et (*ptr).y=90
la flèche est juste une contraction, une facilité d'écriture
*/
/*
#include <stdio.h>
#include <stdlib.h>

int main()
{
struct test{
   int x,y;
} t;
struct test *ptr;

   ptr=&t;
   ptr->x=450;
   ptr->y=90;
   printf("t.x=%d, t.y=%d\n", t.x, t.y);

   return 0;
}
*/
/****************************************************************
4) "crochet" : [] c'est l'opérateur tableau, à partie d'une
adresse de départ il permet d'accéder aux adresses des différents
éléments du tableau

EXEMPLE :

int tab[50];
int*ptr;
    ptr=tab;   // ptr prend l'adresse du tableau cad
               // l'adresse du premier élément du tableau

    for (i=0; i<50; i++)
       ptr[i] = rand()%256;

    // équivalent à :
    for (i=0; i<50; i++)
       *(ptr+i) = rand()%256;

*/
/*
#include <stdio.h>
#include <stdlib.h>

int main()
{
int tab[10];
int*ptr;
int i;

   ptr=tab;
   for (i=0; i<10; i++){
      ptr[i] = rand()%256;
      printf("ptr[%d] = %d\n",i,ptr[i]);
```

```
    }

    // équivalent à :
    for (i=0; i<10; i++){
       *(ptr+i) = ptr[i]+1;
       printf("*(ptr+%d) = %d\n",i,*(ptr+i));
    }
}
*/
```

8. Mise en pratique : base pointeurs

a. Avoir des pointeurs et les manipuler

Exercice 1
Dans un programme, déclarer un int, un double et un float. Leur affecter des valeurs aléatoires entre 600 et 700 avec une valeur décimale. Afficher les valeurs. Modifier les valeurs de chaque variable via son adresse mémoire récupérée dans un pointeur et afficher le nouveau résultat. A l'issue le programme demande s'il faut recommencer ou quitter.

Exercice 2
Dans un programme déclarer deux variables, leur affecter à chacune une valeur et inverser les valeurs sans toucher directement les variables mais en passant par leurs adresses mémoire récupérées dans des pointeurs. Afficher avant et après modification. Le programme quitte si l'utilisateur le demande.

Exercice 3
Soit une structure personne comprenant des informations de nom, prénom, adresse, age, date de naissance, nationalité, métier, hobby. Dans un programme :

- définir le type
- initialiser une structure personne uniquement en passant par l'adresse mémoire
- afficher le résultat
- recommencer ou quitter.

Modifier le programme pour avoir un tableau de nb structures personne. Faire une fonction d'initialisation et initialiser le tableau (de préférence avec des valeurs aléatoires). L'utilisateur peut modifier l'élément de son choix toujours en passant par l'adresse mémoire via un pointeur.

Exercice 4
Dans un programme, soit un tableau d'entiers initialisés à 0 :
l'utilisateur entre le nombre nb de modifications qu'il veut faire
faire ensuite les nb modifications (que vous voulez) sur des éléments sélectionnés au hasard, uniquement si la valeur de l'élément est inférieure à 10. Les modifications sont faites en utilisant l'adresse mémoire de chaque élément via un pointeur d'entier. quitter ou recommencer

b. Tests tableaux / pointeurs

Exercice 5

Dans un programme initialiser un tableau statique de 5 entiers avec des valeurs comprises entre 0 et 255 (à la déclaration). Ensuite récupérez l'adresse du tableau avec un pointeur char* et afficher votre tableau avec le pointeur de char … Que se passe t-il ? Combien de valeurs pouvez-vous afficher à votre avis ? Quelle boucle permet de parcourir tout l'espace mémoire du tableau ?

Exercice 6

Quels résultats fournit ce programme ?

```
#include <stdio.h>

int main()
{
int t[3];
int i, j;
int* ptr;

   for (i=0; j=0; i<3; i++)          // 1
       t[i]=j++ +i;

   for (i=0; i<3; i++)               // 2
       printf("%d ", t[i]);
   putchar('\n');

   for (i=0; i<3; i++)               // 3
       printf("%d ", *(t+i));
   putchar('\n');

   for (ptr=t; ptr<t+3; ptr++)       // 4
       printf("%d ", *ptr);
   putchar('\n');

   for (ptr=t+2; ptr>=t; ptr--)      // 5
       printf("%d ", *ptr);
   putchar('\n');

   return 0;
}
```

Exercice 7

Dans un programme, un tableau de 15 entiers est déclaré mais pas de pointeur. Écrire de deux façons différentes, l'une avec l'opérateur crochet [ ] et l'autre avec l'opérateur étoile * :

- l'initialisation du tableau avec 12 valeurs aléatoires et 3 entrées par l'utilisateur
- l'affichage du tableau
- la recherche du plus grand
- la recherche du plus petit
- l'affichage des résultats

Exercice 8

Dans un programme, un tableau de 10 entiers est déclaré.

- Faire une fonction d'initialisation du tableau en parcourant le tableau avec un pointeur
- Faire une fonction d'affichage du tableau avec parcours par un pointeur
- Quitter ou recommencer

Exercice 9

Dans un programme, une matrice de 20 * 15 entiers est déclarée.
- □ Faire une fonction d'initialisation de la matrice dans laquelle la matrice est parcourue avec un pointeur. Les valeurs seront aléatoires sauf une entrée par l'utilisateur toutes les 100 valeurs.
- □ Faire une fonction d'affichage pareillement parcourue avec un pointeur.
- □ Quitter ou recommencer

Exercice 10
Dans un programme :
- □ saisir une chaine de caractères
- □ faire une fonction d'affichage de la chaine à l'envers en parcourant la chaine avec un pointeur.
- □ quitter ou recommencer

c. Base allocation dynamique

Exercice 11
Dans un programme :
- □ allouez de la mémoire pour un char, un entier, un float
- □ initialiser avec des valeurs entrées par l'utilisateur
- □ affichez
- □ quitter si l'utilisateur le demande (sinon recommencer, attention mémoire)

Exercice 12
Dans un programme :
- □ écrire une fonction d'allocation avec contrôle d'erreur pour un entier
- □ initialiser un tableau de 10 pointeurs d'entier
- □ donner des valeurs décroissantes aux entiers
- □ afficher les valeurs
- □ quitter si l'utilisateur le demande (sinon recommencer, attention mémoire)

Exercice 13
Dans un programme, une entité est définie par un nom, un caractère, un code génétique stocké dans un tableau de 4 entiers (4 séquences), une position, un déplacement, une apparence, une couleur, éventuellement d'autres traits caractéristiques :
- □ définir un type pour l'entité
- □ déclarer deux pointeurs pour deux entités et initialiser avec des valeurs chacune des caractéristiques des entités
- □ comparer les caractéristiques des deux entités et afficher le résultat.
- □ quitter si l'utilisateur le demande ou recommencer, attention à la mémoire.

Exercice 14
Dans un programme, reprendre le type défini pour une entité dans l'exercice 13 et :
- □ déclarer deux pointeurs entités
- □ écrire une fonction d'initialisation et initialiser chaque entité
- □ écrire une fonction d'affichage et afficher chaque entité
- □ écrire une fonction qui affiche le nom de l'entité avec le caractère le plus fort

Exercice 15
Dans un programme, reprendre le type défini pour une entité dans l'exercice 13 et :
- □ déclarer un tableau de pointeurs pour NB_MAX entités

- □ écrire une fonction d'initialisation et initialiser le tableau
- □ écrire une fonction d'affichage et afficher le tableau
- □ écrire une fonction qui affiche le nom de l'entité avec le caractère le plus fort

d. Attention aux erreurs

Exercice 16

Que fait le programme suivant ?, Y a t-il des erreurs, si oui lesquelles ? comment les corriger ? :

```
int main()
{
char*s1;
char s2[80];
int i;

   fgets(s2,100,stdin);
   strcpy(s1, s2);

   s1=s2;
   fgets(s2,100,stdin);
   if (strcmp(s1,s2)==0)
       printf("les deux phrase sont identiques\n");
   else
       printf("elles sont différentes\n");

   s1="bonjour";
   printf("s1 :%s\n",s1);
   strcpy(s2,"il fait beau je prends ma canne et mon chapeau");
   printf("s2 :%s\n",s2);

   while(s2[i]){
       s1[i]=s2[i];
       i++;
   }
   printf("s1 :%s\n",s1);
   printf("s2 :%s\n",s2);
   return 0;
}
```

e. Tableaux de chaines

Exercice 17

Créer un jeu de 52 cartes à partir d'une liste de mots. Au départ, affichez le jeu ordonné par couleurs. Ensuite, faire une fonction de mélange des cartes, faire une fonction de distribution des cartes entre deux joueurs et afficher le jeu de chacun.

Exercice 18

soit une liste de mot :

```
char*liste[ ={"moule","frite","patate","pomme","camembert",
              "gruyère","crêpes", "miel", "cidre","omelette","fin"};
```

Faire un programme qui affiche la liste telle quelle, puis la liste classée en ordre alphabétique. Le classement par ordre alphabétique est fait dans une fonction qui reçoit la liste en argument.

B. Allocation dynamique de tableaux

1. Allouer un tableau avec un pointeur

Voici une allocation dynamique pour un entier :

```
int *ptr;
    ptr=(int*)malloc(sizeof(int) );
```

ce qui est identique à

```
ptr=(int*)malloc(sizeof(int) * 1);
```

Considérons que ptr contient l'adresse du premier élément d'un tableau de 1 élément, cela permet d'écrire indifféremment

```
*ptr=10;
// ou
ptr[0]=10;
```

Les deux notations sont équivalentes.

Et pour avoir plus d'un élément dans le tableau il suffit d'allouer davantage de mémoire, au prorata du nombre souhaité des éléments. Par exemple pour avoir un tableau de 16 entiers :

```
int *ptr
    ptr=(int*)malloc(sizeof(int)*16);
```

Le parcours du tableau est habituel :

```
int i,
    for (i=0; i<16; i++){
        ptr[i]=rand()%100;                  // initialisation
        printf("ptr[%d]=%d",i, ptr[i]);     // affichage
    }
```

Eventuellement la taille du tableau peut être obtenue aléatoirement :

```
int*ptr, taille;
    taille=rand();
    ptr=(int*)malloc(sizeof(int)*taille);
```

Si un dépassement de la mémoire RAM est à craindre au moment de l'allocation il est possible de contrôler le retour de la fonction malloc() qui renvoie NULL si elle n'a pas pu réaliser l'opération demandée. Par exemple :

```
ptr=(int*)malloc(sizeof(int)*taille);
if ( ptr==NULL)
    exit(EXIT_FAILURE);
```

En cas d'erreur, le programme prend fin avec EXIT_FAILURE comme valeur de retour (EXIT_FAILURE vaut 1 et indique au système d'exploitation qu'un problème s'est produit lors de l'exécution du programme).

2. Allouer une matrice avec un pointeur de pointeur

Qu'est ce qu'une matrice ? Une matrice est un tableau de tableaux d'objets d'un type donné. Par exemple une matrice d'entier :

```
int mat[4][7]
```

mat est un tableau de 4 tableaux de 7 entiers. Du fait de l'équivalence tableau-pointeur, la matrice peut être remplacée par un pointeur de tableau, c'est-à-dire par un pointeur de pointeur, ce qui s'écrit :

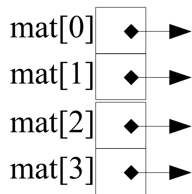
```
int**mat;
```

Le premier niveau de pointeur sert à allouer un tableau de pointeurs. C'est le tableau des lignes et il a la taille du nombre de lignes souhaité. Par exemple, en vue de faire une matrice d'entiers de 4 lignes par 7 colonnes voici la création du tableau de pointeurs pour faire 4 lignes :

```
   1) un tableau de pointeurs

       int**mat;
             mat=(int**)malloc(sizeof(int*)*4);
```

Représenté graphiquement nous obtenons :

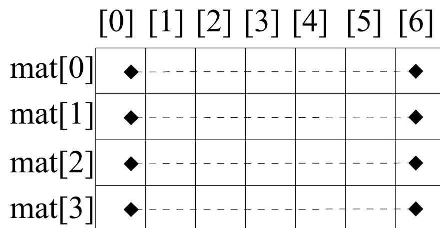


Cet appel de la fonction malloc() alloue une zone mémoire pour 4 pointeurs d'entiers, c'est un tableau de 4 pointeurs d'entiers. Maintenant reste à allouer le tableau correspondant à chaque ligne selon le nombre de colonnes souhaité. S'il y a 7 colonnes, ça donne :

```
   2)chaque pointeur du tableau est alloué en tableau de int

       for (i=0; i<4; i++)
             mat[i]=(int*)malloc(sizeof(int)*7);
```

Représenté graphiquement nous obtenons :



- *Remarque :*

Il est envisageable d'avoir des lignes de tailles différentes mais dans ce cas il faut soit conserver la taille de chaque ligne quelque part soit avoir dans chaque ligne une sentinelle qui indique sa fin ensuite (par exemple le '\0' des chaines de caractères)

Le programme suivant crée une matrice de taille aléatoire et l'initialise avec des valeurs de 0 ou 1 :

```
int main(int argc, char *argv[])
{
int**mat;                                          // 1
int tx,ty,i,j,fin=0;
```

```
    srand(time(NULL));
    do{

        tx=1+rand()%30;                         // 2
        ty=1+rand()%10;
        mat=(int**)malloc(sizeof(int*)*ty);
        for (i=0; i<ty; i++){
             mat[i]=(int*)malloc(sizeof(int)*tx);

             for (j=0; j<tx; j++){
                 mat[i][j]=rand()%2;
                 printf("%2d",mat[i][j]);
             }
        putchar('\n');
        }

        for (i=0; i<ty; i++)                    // 3
        free(mat[i]);
        free(mat);

        printf("\nAutre matrice ? (o/n)\n\n");
        fin=getch();
    }while(fin=='o');

    return 0;
}
```

(1) Déclaration des variables du programme. "mat" pour la matrice est un pointeur de pointeur sur entier, ty tx pour son nombre de lignes et de colonnes.

(2) Initialisation aléatoire des variables tx et ty dans les fourchettes respectives de 1 à 30 et 1 à 10. Et ensuite :

- □ Allocation dynamique du tableau de pointeurs de ty lignes.
- □ Allocation dynamique de chaque ligne du tableau avec tx colonnes

(3) Destruction de matrice, le programme ne va pas utiliser plus cette matrice, l'objectif est uniquement de tester la création dynamique d'une matrice et la mémoire qui vient d'être allouée est libérée juste après la création et l'affichage de la matrice.

Nous pourrions de même évoquer la création dynamique de tableaux à trois dimensions, voir à n dimensions. Le principe d'allocation serait exactement le même. Par exemple voici un ensemble de 10 matrices de 20 lignes par 30 colonnes d'entiers:

```
 int ***tab;
   tab=(int***)malloc(sizeof(int**)*10);
   for (z=0; z<10;z++){
       tab[z]=(int**)malloc(sizeof(int*)*20);
           for (y=0; y<20; y++)
               tab[z][y]=(int*)malloc(sizeof(int)*30);
   }
```

C'est toutefois rare, en général nous trouvons surtout des tableaux à une ou deux dimensions qui fassent l'objet d'une allocation dynamique.

3. Différences entre tableaux statiques et dynamiques

Le pointeur est une variable qui peut recevoir différentes adresses correspondant à différents blocs de mémoire. Un tableau statique est une constante. La taille du bloc correspondant et son adresse sont allouées automatiquement et ne peuvent pas être modifiées ensuite.

De plus l'opérateur sizeof() appliqué à un pointeur donne la taille en octet de la variable pointeur (4 octets) alors que s'il est appliqué à un tableau statique il donne la taille en octets du tableau. Soit *a* un tableau, le nombre de ses éléments peut être retrouvé de la façon suivante :

```
int nb_element = sizeof(a) / sizeof(a[0]);
```

Le tableau (statique) est un type en soi. La conversion d'un tableau en pointeur sur le premier élément du tableau n'a pas lieu lorsqu'une expression de type tableau est l'opérande d'un opérateur unaire comme sizeof. C'est pourquoi sizeof(*a*) donne la taille de *a* et non d'un pointeur sur le premier élément de *a*.

4. Autres fonctions d'allocation dynamique

Outre la fonction malloc() ci-dessus présentée, la librairie C standart <stdlib.h> fournit deux autres fonctions pour allouer de la mémoire dynamiquement, ce sont les fonctions calloc() et realloc().

a. Fonction calloc()

```
void * calloc(size_t nb_element, size_t taille_element);
```

La fonction calloc() alloue une zone mémoire du nombre d'éléments multiplié par la taille d'un élément : nb_element*taille_element. La zone allouée est initialisée avec tous les bits à zéro et la fonction retourne une adresse de type générique void*. Exemple,d'utilisation :

```
int main()
{
int nb,i;
int*tab;
   printf("entrer la taille du tableau:\n");     // 1
   scanf("%d",&nb);
   tab=(int*)calloc(nb,sizeof(int));             // 2
   for (i=0; i<nb; i++){                         // 3
       tab[i]=rand()%256;
       printf("tab[%d]=%d\n",i,tab[i]);
   }
   return 0;
}
```

(1) L'utilisateur est sollicité pour entrer une taille de tableau récupérée avec la fonction scanf()

(2) Allocation dynamique du tableau selon la valeur entrée.

(3) Le tableau est ensuite initialisé avec des valeurs aléatoires comprises entre 0 et 255, chaque valeur et affichée.

b. Fonction realloc()

```
void* realloc(void*ptr, size_t nouvelle_taille);
```

La fonction realloc() modifie la taille de la zone mémoire pointée par ptr. Le pointeur ptr doit soit être NULL, soit contenir l'adresse d'une zone mémoire préalablement allouée. Le paramètre nouvelle_taille donne la nouvelle taille de la zone mémoire. Si elle est plus petite les infos de la partie supprimée sont perdues et celles de la partie gardée sont conservées. Si elle est plus grande toutes les infos de la zone initiale sont préservées et la partie ajoutée n'est pas initialisée (contenu indéterminé). Lorsque l'adresse initiale est à NULL une zone est allouée de taille nouvelle_taille. La fonction retourne un pointeur générique qu'il faut caster en fonction du type souhaité. Exemple d'utilisation :

```
int main()
{
int nb,i,k;
int *tab=NULL;                                    // 1

   for (i=0; i<10; i++){                          // 2
       printf("entrer une nouvelle taille de tableau:\n");
       scanf("%d",&nb);
       tab=(int*)realloc(tab,sizeof(int)*nb);
       for (k=0; k<nb; k++){                      // 3
           if(tab[k]!=k)
               tab[k]=k;
           printf("%d ",tab[k]);
       }
   }
   return 0;
}
```

(1) Déclaration des variables, le pointeur tab est mis à NULL.

(2) Le programme réalloue dix fois la zone mémoire pointée par tab. A chaque fois il demande à l'utilisateur d'entrer une taille, la récupère avec la fonction scanf() et réalloue la zone avec realloc().

(3) Une fois la zone réallouée elle est réinitialisée. Chaque élément du tableau prend la valeur de l'indice correspondant. L'initialisation d'un élément n'a lieu que pour une zone ajoutée et dont les valeurs indéterminées seront probablement différentes des indices du tableau.

5. Mise en pratique : Allocation dynamique

a. Allouer dynamiquement des tableaux

Exercice 1

Dans un programme :

- l'utilisateur entre une taille de tableau d'entiers
- une fonction alloue le tableau
- une fonction initialise le tableau avec des nombres aléatoires croissants

- une fonction affiche le tableau
- quitter ou recommencer (si recommencer libérer la mémoire)

Exercice 2
Considérons la structure t_tableau suivante qui contient l'adresse d'un tableau et le nombre d'éléments du tableau :

```
typedef struct{
    int nb_elem;              // nombre d'éléments
    int*tab;                  // tableau potentiel
}t_tableau;
```

1) Écrire la fonction :

```
t_tableau alloue_tableau (int n);
```

qui crée un tableau de n éléments.

2) Écrire la fonction :

```
void destruct_tableau (t_tableau tab);
```

qui libère la mémoire occupée par un tableau

3) Écrire la fonction :

```
void init_tableau (t_tableau tab);
```

qui initialise un tableau avec des valeurs.

4) Écrire la fonction :

```
void affiche_tableau (t_tableau tab);
```

qui affiche le contenu d'un tableau.

5) Écrire la fonction :

```
t_tableau double_tableau (t_tableau tab);
```

qui crée un nouveau tableau de même taille que tab mais initialisé avec le double des valeurs de tab.

Faire un programme avec un menu qui permet ces différentes opérations.

Exercice 3
Dans un programme,

- écrire une fonction de saisie d'une chaine de caractères ; la fonction n'a pas de paramètre
- écrire une fonction de concaténation de deux chaines de caractères ; la fonction a deux paramètres qui sont les chaines à concaténer et renvoie une troisième chaine qui est la concaténation.
- saisir deux chaines et afficher la concaténation
- quitter ou recommencer (attention à la mémoire).

Exercice 4 (suppose connaissance fichier)
On se propose de réaliser une fonction de chargement d'un fichier texte en mémoire centrale. Le formalisme du fichier est le suivant :

```
- la première ligne donne le nombre d'éléments dans le fichier
- les lignes suivantes contiennent chacune un nombre réel.
Par exemple :
    3
    4.98
    123.76
    45.99
```

- Écrire la fonction de chargement qui prend en paramètre un tableau de la taille exacte
- Écrire une fonction d'affichage du tableau
- Tester dans un programme. Lancer plusieurs fois le programme en modifiant à chaque fois le contenu du fichier texte.

Exercice 5

Dans un programme,

- saisir une phrase
- passer cette phrase à une fonction qui retourne un des mots au hasard de la phrase
- afficher la phrase et le mot sélectionné
- quitter ou recommencer

b. Allouer dynamiquement des matrices

Exercice 6

Dans un programme :

- l'utilisateur entre les deux dimensions d'une matrice d'entiers
- une fonction alloue la matrice
- une fonction initialise la matrice avec des nombres aléatoires croissants
- une fonction affiche la matrice
- quitter ou recommencer (si recommencer libérer la mémoire)

Exercice 7

Considérons la structure t_matrice suivante qui contient l'adresse d'une matrice, le nombre de lignes et le nombre de colonnes :

```
typedef struct{
    int ty, tx;                 // lignes, colonnes
    int**tab;                   // matrice
}t_matrice;
```

1) Écrire la fonction :

```
t_matrice allouer_matrice (int tx, int ty);
```

qui crée une matrice de ty*tx éléments.

2) Écrire la fonction :

```
void destruct_matrice (t_matrice mat);
```

qui libère la mémoire occupée par une matrice

3) Écrire la fonction :

```
void init_matrice (t_matrice mat);
```

qui initialise une matrice avec des valeurs.

4) Écrire la fonction :

```
void affiche_matrice (t_matrice mat);
```

qui affiche le contenu d'une matrice.

5) Écrire la fonction :

```
t_matrice double_matrice (t_matrice mat);
```

qui crée une nouvelle matrice d'une taille double que mat et initialisée dans la partie double avec le double des valeurs de mat, simplement recopiée sinon.

Faire un programme avec un menu qui permet ces différentes opérations.

Exercice 8 (suppose connaissance fichier)
On se propose de réaliser une fonction de chargement d'un fichier texte en mémoire centrale. Le formalisme du fichier est le suivant :

```
- la première ligne donne le nombre de lignes et de colonnes d'une
matrice de char
- ensuite chaque ligne a nb colonne éléments
Par exemple :
    3  4
    abcd
    efgh
    ijkl
```

- □ Écrire la fonction de chargement qui prend en paramètre une matrice de la bonne taille
- □ Écrire une fonction d'affichage de la matrice
- □ Tester dans un programme. Lancer plusieurs fois le programme en modifiant à chaque fois le contenu du fichier texte.

c. Allocation dynamique calloc() et realloc()

Exercice 9
Dans un programme, un agenda est constitué de cellules comprenant nom, prénom, téléphone, adresse.
Première partie :

- □ définir un type pour stocker une cellule
- □ l'utilisateur entre le nombre de cellules qu'il veut intégrer
- □ allouer dynamiquement un tableau pour stocker les cellules. A l'issue toutes les cellules sont à 0.
- □ écrire une fonction d'initialisation d'une cellule et remplir le tableau
- □ afficher le tableau rempli
- □ sauver en binaire, le tableau et sa taille (si connaissance fichiers)

Exercice 10 (suppose connaissance fichier)
Dans un programme, reprendre le type défini dans l'exercice 1. L'objectif cette fois est de pouvoir modifier la base de cellules enregistrées sur fichier binaire, soit en ajouter soit en supprimer. Le programme propose maintenant un menu avec quatre possibilités :

- □ loader : récupérer toute la base dans un tableau de la bonne taille
- □ ajouter : augmenter la taille du tableau et ajouter une cellule
- □ supprimer : choisir la cellule à supprimer, la mettre à la fin et diminuer la taille du tableau
- □ sauvegarder : fichier binaire, tableau et taille.

Exercice 11
Dans un programme, dans la boucle principale, la taille t d'un tableau est donnée au hasard :

- □ une fonction alloue dynamiquement un tableau de t pointeurs d'entiers. La première fois tous les pointeurs sont à NULL c'est à dire en fait ((int*)0) .
- □ une fonction initialise avec des valeurs et une fonction les affiche
- □ le programme quitte ou recommence selon le souhait de l'utilisateur

C. Pointeurs en paramètre de fonction

1. Passage par référence

Les paramètres de fonctions sont des variables locales à la fonction initialisées avec des valeurs au moment de l'appel de la fonction. Lors d'un passage de tableau le paramètre est un pointeur qui prend comme valeur l'adresse du premier élément qui est aussi l'adresse de tout le bloc mémoire du tableau. Ce chapitre a pour objet l'étude du passage d'adresse mémoire pour n'importe quel type de variable et pas seulement les tableaux. C'est ce que l'on appelle un "passage par référence" et il s'agit de la référence à une variable via son adresse mémoire. Nous préciserons également quelques finesses quant au passage des tableaux.

a. Cas général d'une variable quelconque

La fonction modif() a un paramètre auquel elle affecte la valeur 50 :

```
void modif(int e1)
{
   e1=50;
}
```

dans le main() une variable v est initialisée avec la valeurs 10. La fonction modif() est appelée et la valeur de v est affectée à son paramètre ce qui donne :

```
int main()
{
int v=10;
   modif(v);
   printf("v=%d \n", v);
   return 0;
}
```

Qu'imprime la fonction printf() ?

… La variable v, locale au contexte d'appel, vaut toujours 10.

Le paramètre de la fonction est une autre variable, locale à la fonction, à laquelle une valeur est affectée au moment de l'appel. Toutes les modifications faites sur cette variable locale à la fonction dans le corps de la fonction modifient uniquement cette variable locale à la fonction.

En C le passage des arguments se fait *toujours* par valeur et c'est uniquement la valeur des variables qui est affectée aux paramètres des fonctions. Ce n'est pas la variable elle-même qui deviendrait paramètre de la fonction.

Toutefois, si l'on utilise un pointeur comme paramètre de fonction et que l'on passe comme valeur pour le paramètre l'adresse mémoire de la variable il devient possible de modifier la valeur de cette variable en passant par son adresse mémoire. C'est un passage par référence. Par exemple :

```
void modif( int *x)
{
   *x=50;
}
```

La fonction modif() a maintenant comme paramètre un pointeur d'entier. En utilisant l'opérateur * (étoile) la fonction modif() va pouvoir écrire écrire à l'adresse mémoire qui lui est communiquée au moment de l'appel. Notre programme devient :

```
int main()
{
int v=10;

   printf( "v=%d",v);  // v vaut 10
   modif_( &v );       // passage de l'adresse de v au paramètre x
   printf( "v=%d",v);  // v vaut maintenent 50
   return 0;
}
```

b. Exemple pour avoir l'heure

Le return ne permet qu'un seul retour à la fois ce qui n'est pas toujours suffisant et risque de compliquer l'écriture des fonctions. L'intérêt du passage par référence est de pouvoir transformer les paramètres d'entrées en paramètres de sortie. Il y a alors autant de sorties que nécessaire. Voici par exemple une fonction qui retourne le temps en heure, minute et seconde.

Le temps universel est donné par la fonction :

```
time_t  time(time_t*tmps) ;
```

Cette fonction retourne la valeur du temps selon un codage interne au système ou -1 si impossibilité. Cette valeur est également recopiée à l'adresse du paramètre sauf s'il est mis à NULL. Ce temps interne au système doit ensuite être converti par exemple avec la fonction :

```
struct tm* localtime(const time_t*tu)
```

Cette fonction convertit la valeur de temps interne, obtenue avec time(), dans un format de temps exploitable nommé temps externe et décrit par la structure tm présentée ci-après. Le temps interne est donné en p1 et l'adresse d'une structure tm allouée dynamiquement et correctement initialisée est retournée. Une structure tm comprend :

```
struct tm{

   int tm_year; // nombre d'années écoulées depuis 1900
   int tm_mon;  // nombre de mois écoulés depuis le début de
                // l'année
   int tm_mday; // jour du mois de 1 à 31
   int tm_yday; // jour de l'année de 0 à 365
   int tm_hour; // heures depuis le début du jour 0 à 23
   int tm_min;  // minutes de 0 à 59 depuis le début de l'heure
                // courante
   int tm_sec;  // secondes depuis début minute courante 0 à 59
};
```

Nous pouvons maintenant écrire notre fonction pour avoir l'heure :

```
void hms(int *h, int *m, int*s)
{
time_t tu;              // pour récup temps universel (interne)
struct tm *p;    // pour récup format lisible
```

```
   time(&tu);  // passage par référence variable tu
   p = localtime(&tu); // passage adresse de tu juste pour lecture
                       // mais retour d'une adresse de structure
                      // obtenue dynamiquement dans la fonction

   *h = p->tm_hour;     // écriture à l'adresse h de l'heure
   *m = p->tm_min;      // écriture à l'adresse m des minutes
   *s = p->tm_sec;      // écriture à l'adresse s des secondes
}
```

Et la tester dans un main

```
int main()
{
 int h, m, s;

          hms(&h, &m, &s);
          printf("%d : %d : %d\n",h,m,s);
          return 0;
}
```

c. Passage par référence d'une structure

Reprenons ici notre structure t_troll

```
typedef struct troll{
    int x,y;
    int color;

}t_troll;
```

nous pouvons avoir une fonction d'initialisation avec en paramètre un pointeur t_troll* :

```
void init(t_troll*p)
{
    p->x=rand()%800;
    p->y=rand()%600;
    p->color=rand()%256;
}
```

Et quelque part un appel :

```
t_troll t;
    init(&t);    // passage de l'adresse d'une struct t_troll
```

Dans le cas de notre tableau de pointeur il ne faut pas oublier d'allouer les pointeurs. En effet, écrire par exemple :

```
t_troll* ALL[10];
int i;
    for (i=0; i<10; i++){
        init(ALL[i]);    // ERREUR!  ALL[i] NON ALLOUE
    }
```

provoque une erreur à l'exécution, ALL[i] ne disposant pas d'une adresse réservée. Il faut donc avoir ajouté précédemment l'allocation mémoire (soit au même endroit soit ailleurs dans le programme) :

```
    for (i=0; i<10; i++){
        ALL[i]=(t_troll*)malloc(sizeof(t_troll));
        init(ALL[i]);    // OK,  ALL[i] ALLOUE
    }
```

d. Passage par référence d'une variable pointeur

Le pointeur est une variable comme les autres et à ce titre il est possible de passer un pointeur par référence c'est à dire de passer comme valeur au paramètre l'adresse d'une variable pointeur. L'adresse d'une variable pointeur c'est un pointeur … de pointeur. Voici par exemple une fonction d'allocation de tableau de float qui n'utilise pas le retour :

```
void alloue(float**f,int taille)
{
   *f=(float*)malloc(sizeof(float)*taille);
}
```

et l'appel dans un main()

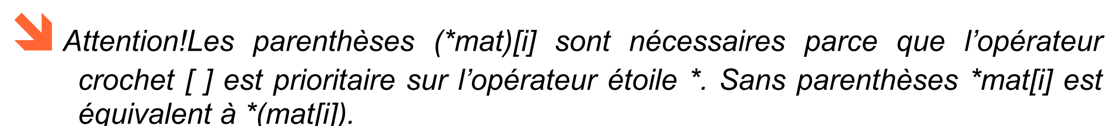
```
int main()
{
 float*tab;
 int i;
   alloue(&tab, 10);
   for (i=0; i<10; i++){
       tab[i]= (rand()%10000)/100.0;
       printf("%f ",tab[i]);
   }
   putchar('\n');

   return 0;
}
```

Ce sont exactement les mêmes manipulations que pour pour toutes les variables sauf que la variable en question est un pointeur et s'utilise ensuite comme un pointeur. Ainsi dans la fonction alloue() l'expression *f est de type float* ce qui permet une allocation dynamique de tableau de float.

Autre exemple, nous voulons une fonction d'allocation de matrices d'entiers qui n'utilise pas le retour. Une matrice c'est un pointeur de pointeur int** et l'adresse d'un pointeur de pointeur c'est un pointeur... de pointeur de pointeur, un int*** :

```
void alloue(int***m,int t1, int t2)
{
int i;
   *m=(int**)malloc(sizeof(int*)*t1);
   for (i=0; i<t1; i++)
      (*m)[i]=(int*)malloc(sizeof(int)*t2);
}
```



*Attention!Les parenthèses (*mat)[i] sont nécessaires parce que l'opérateur crochet [ ] est prioritaire sur l'opérateur étoile *. Sans parenthèses *mat[i] est équivalent à *(mat[i]).*

Pour tester l'allocation dynamique réalisée voici une fonction d'initialisation et une fonction d'affichage :

```
void init(int**mat, int t1, int t2)
{
int i, j;
   for (i=0; i<t1; i++)
```

```
      for (j=0; j<t2; j++)
         mat[i][j]=rand()%100;
}

void affiche(int**mat, int t1, int t2)
{
int i, j;
   for (i=0; i<t1; i++){
      for (j=0; j<t2; j++)
         printf("%3d",mat[i][j]);
      putchar('\n');
   }
}
```

et les appels dans un main()

```
int main()
{
int**mat;

   alloue(&mat,5,10); // à l'adresse de mat passée par référence
   init(mat,5,10);   // à la valeur de mat (adresse de la matrice)
   affiche(mat,5,10);
   return 0;
}
```

Dernier exemple reprenons notre structure t_troll

```
typedef struct troll{
    int x,y;
    int color;

}t_troll;
```

et modifions la fonction d'initialisation de façon à y intégrer l'allocation mémoire du pointeur en passant le pointeur par référence :

```
void init(t_troll**p)
{
    *p=(t_troll*)malloc(sizeof(t_troll));
    (*p)->x=rand()%800;
    (*p)->y=rand()%600;
    (*p)->color=rand()%256;
}
```

l'expression *p est est de type t_troll* et les parenthèse sont nécessaire autour de *p parce que la flèche est prioritaire sur l'étoile. Sans parenthèse

```
*p->x est équivalent à *(p->x)
```

ce qui suppose que le champ x est de type pointeur (et attention au bug si c'est effectivement le cas).

L'initialisation d'un tableau de pointeurs t_troll* peut faire l'objet d'une fonction à part :

```
void initialisation(t_troll* t[])
{
int i;
    for (i=0; i<NBMAX; i++)
       init(&t[i]);     // passage de l'adresse du pointeur i dans
                        // le tableau
}
```

Voici le programme complet :

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

#define NBMAX   100
typedef struct troll{
    int x,y;
    int color;

}t_troll;

void init(t_troll**p)
{
    *p=(t_troll*)malloc(sizeof(t_troll));
    (*p)->x=rand()%800;
    (*p)->y=rand()%600;
    (*p)->color=rand()%256;
}

void initialisation(t_troll* t[])
{
int i;
    for (i=0; i<NBMAX; i++)
       init(&t[i]);
}

void affiche(t_troll*t[])
{
int i;
    for (i =0; i<NBMAX; i++){
       printf("%4d %4d %4d\n",t[i]->x,t[i]->y,t[i]->color);
    }
}
int main()
{
t_troll* ALL[NBMAX];

    initialisation(ALL);
    affiche(ALL);
    return 0;
}
```

2. Tableaux dynamiques en paramètre

Dans le cas d'un tableau statique en paramètre de fonction, seule la première dimension du tableau est convertie en pointeur. Par exemple s'il s'agit d'une matrice d'entiers, un tableau à deux dimensions, la matrice est convertie en pointeur de tableaux d'entiers, et pour cette raison la taille de la seconde dimension du tableau doit être spécifiée. Ce fonctionnement est identique quelque soit le nombre des dimensions d'un tableau statique en paramètre : seule la première dimension est convertie en pointeur et les autres dimensions constituent le type du pointeur, leurs tailles doivent être spécifiées.

Voici un petit test pour voir les compatibilité et incompatibilité entre trois formes de tableaux assez proches quoique différentes :

```
#define TY         3    // nombre de lignes
#define TX         8    // nombre de colonnes
```

```
char mat1[TY][TX];      // matrice statique de char
char*mat2[TY];          // tableau statique de pointeurs de char
char**mat3;             // matrice dynamique de char OU
                        // tableau dynamique de pointeurs de char
```

Ecrire trois fonctions identiques d'affichage avec chacune un tableau différent en paramètre :

```
void test1      ( char**tab )
void test2 ( char* tab[ ] )
void test3 ( char tab[ ][TX] )

/* contenu des trois :
{
int x,y;
    putchar('\n');
    for (y=0;y<TY;y++){
        putchar('\t');
        for (x=0; x<TX; x++)
            putchar(tab[y][x]);
        putchar('\n');
    }
    putchar('\n');
}
```

Comment se comporte la fonction test1 avec le paramètres mat1, mat2 et mat3 ?

Comment se comporte la fonction test2 avec le paramètres mat1, mat2 et mat3 ?

Comment se comporte la fonction test3 avec le paramètres mat1, mat2 et mat3 ?

Nous obtenons les compatibilités et incompatibilités suivantes :

Types passés aux paramètres	**Types des paramètre**		
	(char**tab)	(char* tab[ ])	(char tab[ ][TX])
char mat1[TY][TX]	NO	NO	OK
char*mat2[TY]	OK	OK	NO
char**mat3	OK	OK	NO

Le programme ci-dessous permet de le vérifier :

```
int main(int argc, char *argv[])
{
// les déclarations des tableaux
char mat1[TY][TX];
char*mat2[TY];
char**mat3;
int y,x;
```

```
    // les tableaux sont alloués si nécessaire et initialisés
    // avec les mêmes valeurs
    for (y=0; y<TY; y++)
       for (x=0; x<TX; x++)
          mat1[y][x]='0'+x;

    for (y=0; y<TY; y++){
       mat2[y]=(char*)malloc(sizeof(char)*TX);
       for (x=0; x<TX; x++)
          mat2[y][x]='0'+x;
    }

    mat3=(char**)malloc(sizeof(char*)*TY);
    for (y=0;y<TY;y++){
       mat3[y]=(char*)malloc(sizeof(char)*TX);
       for (x=0; x<TX; x++)
          mat3[y][x]='0'+x;
    }

    // première série de test avec mat1 en paramètre pour chacune
    //des fonctions
    printf("TEST 1  Passage de :  char mat1[TY][TX] \n\n");

    printf("appel de la fonction f(char**tab)    :  "
           "Warming et plante\n");
    //test1(mat1);
    putchar('\n');
    printf("appel de la fonction f(char* tab[])  :  "
           "Warming et plante\n");
    //  test2(mat1);
    putchar('\n');
    printf("appel de la fonction f(char tab[][TX]): ok\n");
    test3(mat1);

    // série 2 avec mat2
    printf("TEST 2  Passage de :  char *mat2[TY] \n\n");
    printf("appel de la fonction f(char**tab)          : ok\n");
    test1(mat2);
    printf("appel de la fonction f(char* tab[])     : ok\n");
    test2(mat2);
    printf("appel de la fonction f(char tab[][TX])  : "
           "Warming et resultats incertains\n");
    test3(mat2);

    // série 3 avec mat3
    printf("TEST 3  Passage de :  char**mat3 \n\n");
    printf("appel de la fonction f(char**tab)       :   ok\n");
    test1(mat3);
    printf("   appel de la fonction f(char* tab[])  :   ok\n");
    test2(mat3);
    printf("   appel de la fonction f(char tab[][TX]):   "
           "Warming et resultats incertains\n");
    test3(mat3);
    return 0;
}
```

3. Mise en pratique : Passage par référence

a. Passage par référence, base

Exercice 1
Dans un programme, une fonction initialise deux entiers et un réel avec des valeurs aléatoires. Une autre fonction affiche les valeurs obtenues Le programme quitte sur commande de l'utilisateur

Exercice 2
Dans un programme une fonction retourne le quotient et le reste de la division d'un entier p par un entier q ; p et q sont obtenus soit de façon aléatoire soit entrés par l'utilisateur. Le programme quitte sur commande de l'utilisateur.

Exercice 3
Dans une fonction les valeurs de deux variables passées par référence sont échangées. Faire un programme avec

- saisie des valeurs et affichage
- échange
- affichage du résultat
- quitter ou recommencer

Exercice 4
Soit une structure comprenant une position (réel), un déplacement (réel), une lettre (entier) et une couleur (entier). dans un programme :

- définir un type et déclarer deux structures
- écrire une fonction qui permet d'initialiser deux structures en un seul appel (les structures sont passées par référence).
- écrire une fonction qui permet d'échanger les contenus des deux structures.
- quitter ou recommencer.

Exercice 5
Dans un programme :

- écrire une fonction d'allocation dynamique d'un tableau de n entiers (la taille est obtenue soit via l'utilisateur, soit de façon aléatoire)
- écrire une fonction d'initialisation du tableau avec des valeurs comprises entre un seuil bas et un seuil haut fournis en paramètre de la fonction.
- écrire une fonction de récupération des seuils bas et haut
- écrire une fonction d'affichage d'un tableau de n éléments
- écrire une fonction qui permet au contexte d'appel de déterminer les valeurs maximale et minimale d'un tableau d'entiers de taille n passé en argument. La fonction ne retourne rien mais les valeurs doivent pouvoir être récupérées dans le contexte d'appel sans utilisation de variable globale.

Tester dans un programme qui s'arrête lorsque l'utilisateur le demande.

b. Passage par référence, opérateurs bit à bit

Exercice 6

Vous devez réaliser en C une librairie pour programmer des micro-contrôleurs. Voici quelques unes des fonctions à faire :

bit_clear()	`Syntaxe : bit_clear(var,bit)`
	`rôle    : mettre à 0 le bit "bit de la variable "var"`
	`exemple : bit_clear(a, 3) ;  // met à 0 le bit 3 de a`

bit_set()	`Syntaxe : bit_set(var, bit)`

	Rôle : mettre à 1 le bit "bit de la variable "var" Exemple : bit_set(a,5) ; met à 1 le bit 5 de a
bit_test()	Syntaxe : bit_test(var, bit) Rôle : test l'état du bit "bit" de la variable "var" Exemple : a=2 ; bit_test(a, 2) ; retourne 1, bit 2 de a à 1
get_oct()	Syntaxe : get_oct(var,oct) Rôle : retourne la valeur de l'octet "oct" de la variable "var" Exemple : a=0xFF00 ; get_oct(a,2) ; retourne 255
set_oct()	Syntaxe : set_oct(adresse,oct,val) Rôle : met à "val" l'octet "oct" de la variable "var" Exemple : a=0 ; set_oct(&a,0,0xF) ; donne la valeur décimale 15 à l'octet 0 de a
set_hight()	Syntaxe : set_hight(adresse,oct) Rôle : met à 1 les bits de l'octet "oct" Exemple : a=0 set_hight(&a,3) ; met à 1 les 8 bits du dernier octet de a
set_low()	Syntaxe : set_low(adresse,oct) Rôle : met à 0 les bits de l'octet "oct" Exemple : a=7654 set_hight(&a,0) ; met à 0 les 8 bits du premier octet de a
rotate_left()	Syntaxe : rotate_left(adress,oct, n) Rôle : rotation à gauche de n positions de l'octet "oct" Exemple : a=512 ; rotate_left(&a,2,3) ; fait tourner de trois bits vers gauche l'octet 2 de a
rotate_right()	Syntaxe : rotate_right(adress,oct, n) Rôle : rotation à droite de n positions de l'octet "oct" Exemple : a=512 ; rotate_right(&a,2,3) ; fait tourner de trois bits vers droite l'octet 2 de a

c. Passage de pointeurs par référence

Exercice 6

Dans un programme, dans la boucle principale :

- les tailles d'un tableau de floats et la taille d'un tableau d'entiers sont données au hasard

- □ une fonction alloue dynamiquement les deux tableaux à la fois
- □ une fonction initialise les deux tableaux
- □ une fonction affiche les deux tableaux
- □ quitter ou recommencer, attention à la mémoire.

Exercice 7
dans un programme
- □ l'utilisateur entre la taille d'une matrice
- □ la matrice est allouée dynamiquement dans une procédure (pas de retour)
- □ une fonction initialise la matrice avec des valeurs
- □ une fonction affiche la matrice
- □ quitter ou recommencer, attention mémoire

Exercice 8
Dans une programme, quatre chaines de caractères sont déclarées sous la forme :
char*s1,*s2,*s3,*s4;
- □ une fonction permet de saisir les quatre chaines de caractères à la fois
- □ une fonction permet de les afficher une par une
- □ quitter ou recommencer

Exercice 9
Dans un jeu il y a des trolls avec une position (x,y) et une couleur.
- □ définir une structure t_troll *(cours p.104 et p.115 pour typedef)*
- □ écrire une fonction d'allocation dynamique de tableau de t_troll à 5 dimensions avec utilisation de return.
- □ Écrire une fonction d'allocation dynamique de tableau de t_troll à 5 dimensions sans utilisation du return.

Ensuite toujours sans return décomposer l'allocation en écrivant une fonction pour chaque dimension (pour chaque nouvelle dimension appeler la fonction de la dimension précédente, comme une sorte de cascade) :
- □ Ecrire une fonction qui alloue un tableau à une dimension de t_troll dont la taille est donnée en paramètre.
- □ Ecrire une fonction qui alloue un tableau à deux dimensions dont les tailles sont données en paramètre.
- □ Ecrire une fonction qui alloue un tableau à trois dimensions dont les tailles sont données en paramètre.
- □ Ecrire une fonction qui alloue un tableau à quatre dimensions dont les tailles
- □ sont données en paramètre.
- □ Ecrire une fonction qui alloue un tableau à cinq dimensions dont les tailles sont données en paramètre.

Dans un main donnez un exemple d'appel qui permet d'avoir un tableau de t_troll à 5 dimensions chacune entrée par l'utilisateur. Tester les programme, ajouter initialisation et affichage.

d. Passage de tableaux dynamiques

Exercice 10
Dans un programme un tableau dynamique est déclaré :
```
int *tab;
```

D'après ces prototypes :

```
void      alloue1      (int**t, int taille);
void      alloue2      (int taille);
int*      alloue3      (int taille);
void      initialise1  (int t[], int taille);
void      initialise2  (int**t);
void      initialise3  (int*t);
int*      initialise4  (void);
```

Quelles fonctions peuvent utiliser tab comme argument ? Écrire les fonctions que vous avez désignées comme bonnes et faire un programme de test. Donnez les raisons pour lesquelles vous avez rejeté les autres.

Exercice 11

Dans un programme une matrice dynamique est déclarée :

```
int **mat;
```

D'après ces prototypes :

```
void      alloue1      (int***m, int lig, int col);
int**     alloue2      (int lig, int col);
int*      alloue3      (int*m[],int lig,int col);
void      initialise1  (int m[][]);
void      initialise2  (int**m, int lig, int col);
void      initialise3  (int*t, int lig);
int**     initialise4  (void);
```

Quelles fonctions peuvent utiliser tab comme argument ? Ecrire les fonctions que vous avez désignées comme bonnes et faire un programme de test. Donnez les raisons pour lesquelles vous avez rejeté les autres.

D. Fichiers (type FILE*)

1. Base fichier

a. Le type FILE*

Dans un programme C un fichier est toujours une structure de type FILE manipulée via son adresse par un pointeur FILE*. Pour utiliser un fichier dans un programme ou une fonction du programme la première chose à faire est de déclarer un FILE* :

```
int main()
{
FILE*f;
    (...)
   return 0;
}
```

b. Ouverture et fermeture d'un fichier

Dans la librairie standard stdio.h il y a la fonction :

```
FILE* fopen( const char* name ,const char* mode);
```

pour ouvrir un fichier existant ou créer un fichier inexistant. Cette fonction ouvre le fichier dont le nom comprenant le chemin d'accès est spécifié au premier paramètre et selon le mode donné par le second paramètre. Les deux sont des chaines de

caractères. Il y a six modes possibles : r, w, a, r+, w+, a+ et le caractère "b" ajouté ensuite permet de sélectionner une entrée/sortie en mode binaire. Voici un tableau récapitulatif des différents modes possibles :

Mode	Accès	Positionnement en écriture	Si le fichier existe	Si le fichier n'existe pas
r	Lecture	au début	ouverture	erreur
w	Ecriture	au début	initialisation	création
a	Ecriture	à la fin	ouverture	création
r+	Lecture	au début	ouverture	erreur
w+	et	au début	initialisation	création
a+	écriture	à la fin	ouverture	création
Suffixe b : à jouter pour les entrées sorties binaires.				

Attention, l'initialisation d'un fichier existant signifie qu'il est ramené à une taille nulle, c'est-à-dire que toutes les données qu'il contient sont perdues.

Pour refermer un fichier précédemment ouvert, la librairie stdio.h propose la fonction

```
int   fclose(FILE*);
```

Cette fonction referme le fichier indiqué via le pointeur au paramètre p. Elle retourne EOF sur erreur et zéro sinon. EOF est une valeur sentinelle qui indique la fin du fichier.

Exemple d'ouverture fermeture de fichier en mode binaire :

```
int main()
{
FILE*f;

    if (f=fopen("test.bin","rb")){
        printf("le fichier binaire test.bin existe\n");
        fclose(f);
    }
    else{
        printf("le fichier binaire test.bin n'existe pas\n");
        if (f=fopen("test.bin","wb"))
        {
            printf("le fichier binaire test.bin a ete cree\n");
            fclose(f);
        }
        else
            printf("erreur création fichier binaire text.bin\n");
    }
    return 0;
}
```

Lors du premier lancement ce programme crée le fichier binaire "test.bin" localisé dans le même répertoire que celui du programme (ou au niveau du projet pendant le travail avec un compilateur).

c. Spécifier un chemin d'accès

Le nom du fichier correspond éventuellement au chemin à parcourir pour y accéder soit à partir de la racine d'un disque C:/(chemin absolu), soit à partir du répertoire dans lequel se trouve le programme (chemin relatif). Dans le cas d'un chemin relatif, le point suivi d'un slash indique que l'on descend à partir du répertoire courant dans les sous répertoires. Par exemple ". /exo 1/part 4/test.txt" signifie à partir du répertoire courant, dans le dossier exo1, dans le dossier part 4, le fichier test.txt. A l'inverse on remonte à partir du répertoire courant avec deux caractères points juxtaposés. Par exemple : "../../" permet de remonter de deux niveaux dans l'arborescence à partir du répertoire courant. Il est bien sur possible de combiner les deux, de remonter d'abord d'un ou plusieurs niveaux et ensuite de redescendre pour atteindre un fichier. Par exemple "../.. /exo 1/part 4/test.txt" signifie de remonter de deux niveaux et de redescendre via le dossier exo 1 puis part 4 pour atteindre le fichier test.txt. Attention les dossiers doivent exister et se trouver effectivement aux bons endroits sinon le chemin d'accès au fichier n'est pas valable.

A noter la macro constante FILENAME_MAX qui définit la taille maximum du plus long nom possible de fichier (chemin compris) supporté par le système.

Exemple chemin relatif :

```
int main(int argc, char *argv[])
{
FILE*f;
   if ( (f=fopen("../../test.txt","w"))!=NULL)
       printf("le fichier est créé 2 repertoires au dessus du"
              "repertoire ou se trouve le programme\n");
   else
       printf("erreur création fichier\n");
   return 0;
}
```

2. Fichiers binaires

a. Écriture et lecture en mode binaire

L'utilisation de fichiers binaires repose sur deux fonctions uniquement. Une pour l'écriture et une pour la lecture. Ces deux fonctions sont dans la librairie standard stdio.h

Pour écrire (save) des données :

```
size_t          fwrite ( const void*, size_t, size_t, FILE* );
```

Cette fonction écrit sur le fichier spécifié en p4, p3 objets de taille p2 en octet, rangés à partir de l'adresse p1. Elle retourne le nombre d'objets effectivement écrits.

Pour lire (load) les données d'un fichier :

```
size_t          fread ( void*, size_t, size_t, FILE*);
```

Cette fonction lit au plus p3 objets de taille p2 en octet sur le fichier spécifié en p4 et les copie à l'adresse p1. Elle retourne le nombre d'objets effectivement lus

Exemple d'utilisation :

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

```
#define NB_POINT    10

typedef struct point{
    int x,y;
}t_point;

void    init        (t_point t[]);
void    affiche     (t_point t[]);

int main()
{
FILE*f;
t_point data[NB_POINT];
t_point recup[NB_POINT];

   init(data);
   affiche(data);
   printf("-----------------------------\n");

   if (f=fopen("test.bin","wb+")){
       fwrite(data,sizeof(data),1,f);
       // ramener le curseur écriture/lecture au début du fichier
       rewind(f);

       if (fread(recup,sizeof(t_point),NB_POINT,f)== NB_POINT)
           affiche(recup);
       else
           printf("erreur lecture fichier binaire\n");
   }
   else
        printf("erreur création fichier binaire\n");

   return 0;
}

void init(t_point t[])
{
int i;
    for (i=0; i<NB_POINT; i++){
        t[i].x=rand()%100;
        t[i].y=rand()%100;
    }
}

void affiche(t_point t[])
{
int i;
    for (i=0; i<NB_POINT; i++)
        printf("t[%3d].x=%3d,.y=%3d\n",i,t[i].x,t[i].y);
}
```

b. Détecter la fin d'un fichier binaire

Une première méthode consiste à relever chaque élément du fichier un par un jusqu'à ce que fread() renvoie 0, ce qui signifie en principe la fin du fichier (sauf une erreur entre deux).

Par exemple une boucle comme :

```
t_point recup[NB_POINT*2];
int dernier=0;

   (...)
   while(fread(&recup[dernier++],sizeof(t_point),1,f))
       ;
```

Attention toutefois à ne pas déborder la taille du tableau qui réceptionne les données.

Une autre méthode consiste à utiliser la fonction de la librairie standard stdio.h :

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

Cette fonction retourne vrai si le la position courante dans le fichier passé en paramètre est à la fin et faux sinon. Peut-être utilisée pour les deux modes, texte et binaire.

Par exemple :

```
while(!feof(f))
    fread(&recup[dernier++],sizeof(t_point),1,f);
```

c. Déplacements dans un fichier

Pour des accès directs à n'importe quel endroit du fichier il y a essentiellement trois fonctions prévues dans la librairie standard stdio.h.

```
int        fseek (FILE*, long, int);
```

Cette fonction permet de modifier la position courante dans le fichier spécifié en p1 à partir du positionnement qui est spécifié en p3 de la façon suivante :

```
SEEK_SET         à partir du début du fichier
SEEK_CUR         à partir de la position courante
SEEK_END         à partir de la fin.
```

Le second paramètre p2 indique un déplacement en octets. Par exemple l'appel :

```
fseek (monFichier, 16, SEEK_SET);
```

déplace la position courante de 16 octets (16 caractères) à partir du début du fichier. En cas d'erreur fseek renvoie une valeur négative et 0 si pas d'erreur.

```
long       ftell (FILE*);
```

Cette fonction retourne la position courante associée au fichier passé en paramètre. Elle échoue si la taille à retourner est supérieure à LONG_MAX (Dans ce cas il faut utiliser la fonction fgetpos()).

```
void       rewind ( FILE*);
```

Cette fonction ramène la position courante au début du fichier spécifié en paramètre. Soit un fichier f ouvert, l'appel

```
rewind(f); // est équivalent à fseek(f,0 SEEK_SET);
```

3. Écriture et lecture en mode texte

Certaines des fonctions présentées dans ce module sont également présentées dans le module sur les chaînes de caractères. Elles y sont utilisées avec les fichiers

d'entrée-sortie standards stdin et stdout. Il s'agit maintenant de les employer plus largement avec toutes sortes de fichiers créés dans un programme. Il y a dans ce module d'autres exemples d'utilisation.

a. Détecter la fin d'un fichier, EOF et feof()

En mode texte surtout, détecter la fin d'un fichier est souvent nécessaire. Il y a deux méthodes. La première du fait de la valeur EOF et la seconde avec la fonction feof().

```
Le caractère de fin de fichier EOF -------   <stdio.h>
```

La fin des fichiers texte et binaire est marquée par une valeur définie par une macro constante EOF définie dans stdio.h. Cette valeur vaut en général -1 mais elle peut varier d'un environnement à l'autre. Dans les tests qui l'utilisent il est plus sûr de toujours utiliser EOF plutôt que -1.

La plupart des fonctions du mode texte utilisent la valeur EOF pour signifier la fin du fichier. Cette valeur est aussi utilisée comme retour d'erreur, par exemple pour la fonction fclose().

```
int        feof ( FILE*) -------      <stdio.h>
```

Cette fonction retourne vrai si le la position courante dans le fichier passé en paramètre est à la fin et faux sinon. Peut-être utilisée pour les deux modes, texte et binaire.

b. Lecture / écriture de caractères

```
int        fgetc  ( FILE*)     -------      <stdio.h>
```

Cette fonction retourne un caractère lu sur le fichier spécifié en paramètre et EOF en cas d'erreur où de détection de la fin du fichier. La macro *int getc(FILE*)* en est une autre forme équivalente.

```
int        fputc ( int, FILE*) -------      <stdio.h>
```

Cette fonction écrit sur le fichier spécifié en p2 le caractère spécifié en p1. Elle retourne le caractère p1, en cas d'erreur elle retourne EOF.

Exemple d'utilisation :

Le programme ci-dessous ouvre en lecture seule le fichier entree.txt. Ce fichier est sensé exister et se trouver dans le même répertoire que le programme. Pour les besoins du test il doit également contenir du texte. Un deuxième fichier sortie.txt est ouvert ou créé s'il n'existe pas encore, en écriture seule. Caractère par caractère le contenu d'entrée.txt est récupéré et copié sur sortie.txt et également sur stdout afin de vérifier visuellement le fonctionnement :

```
#include <stdio.h>
#include <stdlib.h>
#define ERREUR(msg){\
   printf("%s\n",msg);\
   system("PAUSE");\
   exit(EXIT_FAILURE);\
}
```

```
int main(int argc, char *argv[])
{
FILE *in, *out;
int c;

   // ouverture fichier entree en lecture seule
   in=fopen("entree.txt","r");
   if (in==NULL)
       ERREUR("ouverture du fichier entree.txt");

   // ouverture sortie en écriture seule
   if ( (out=fopen("sortie.txt","w"))==NULL)
       ERREUR("ouverture du fichier sortie.txt");

   //lecture écriture caractère par caractère
   while ( (c=fgetc(in))!=EOF){
       fputc(c,out);
       fputc(c,stdout);   // contrôle dans la fenètre console
   }
   fclose(in);
   fclose(out);
   return 0;
}
```

c. Lecture / écriture de chaînes

```
char* fgets (char*, int, FILE*)  -------   <stdio.h>
```

Cette fonction lit une chaîne entrée sur le fichier spécifié au paramètre p3 (dans le cas de l'entrée standard ce fichier sera stdin) jusqu'à un '\n' ou jusqu'à ce que p2-1 caractères aient été lus, le '\n' final compris. Tous les caractères lus sont placés à partir de l'adresse spécifiée au paramètre p1. Un '\0' est ajouté comme dernier caractère après le \n dans le cas d'une lecture sur l'entrée standard stdin. La fonction retourne l'adresse p1 ou NULL en cas d'erreur.

```
int fputs(char*, FILE*)      -------      <stdio.h>
```

Cette fonction permet d'écrire (copier) la chaîne passée au paramètre p1 dans le fichier passé au paramètre p2. Elle retourne EOF si une erreur se produit.

Exemple d'utilisation :

Idem programme précédent mais chaîne de caractères par chaîne de caractères

```
#define ERREUR(msg){\
   printf("%s\n",msg);\
   system("PAUSE");\
   exit(EXIT_FAILURE);\
}

int main(int argc, char *argv[])
{
FILE *in, *out;
char buf[1000];

   // ouverture fichier entree en lecture seule
   in=fopen("entree.txt","r");
   if (in==NULL)
```

```
        ERREUR("ouverture du fichier entree.txt");

    // ouverture sortie en écriture seule
    if ( (out=fopen("sortie.txt","w"))==NULL)
        ERREUR("ouverture du fichier sortie.txt");

    //lecture écriture avec chaînes de caractères
    while ( fgets(buf,1000,in)!=NULL){
        fputs(buf,out);
        fputs(buf,stdout);
    }
    fclose(in) ;
    fclose(out) ;
    return 0;
}
```

d. Lecture / écriture formatées

```
int        fscanf ( FILE*, const char*, …)  -------      <stdio.h>
```

Cette fonction analyse une suite de caractères lus sur le fichier spécifié en p1. Elle la découpe en "token" et convertit chaque token en une valeur. Le découpage et la conversion sont effectués selon la spécification de forma p2. Pour chaque spécification de conversion que contient celle-ci, il doit exister un paramètre, passé par référence, dans la liste des paramètres éventuels à la suite de p2. Chaque valeur résultant d'une conversion est copiée à l'adresse du paramètre correspondant. La fonction retourne le nombre de valeurs correctes trouvées compte tenu de l'adéquation entre les formats et les références de paramètre prévues.

```
int        fprintf ( FILE* const char*, …)  -------      <stdio.h>
```

Cette fonction formate la liste des paramètres éventuels à la suite de p2 selon la spécification de format p2. La chaîne de caractères résultante est écrite dans le fichier spécifié a paramètre p1.

<u>Exemple d'utilisation :</u>

Même principe que dans les programmes précédents mais le contenu du fichier entree.txt contient maintenant trois nombres ainsi qu'une suite de mots pour tester les formats %d et %s. Voici ce contenu : 10, 15, 4, toto fait des courses

Tout ce qui est récupéré dans le fichier en entrée est recopié dans le fichier en sortie et également dans le fichier standard stdout d'affichage console.

```
#define ERREUR(msg){\
    printf("%s\n",msg);\
    system("PAUSE");\
    exit(EXIT_FAILURE);\
}

int main(int argc, char *argv[])
{
FILE *in, *out;
int a,b,nb,i;
char buf[1000] ;

    // ouverture fichier entree en lecture seule
```

```
   in=fopen("entree.txt","r");
   if (in==NULL)
       ERREUR("ouverture du fichier entree.txt");


   // ouverture sortie en écriture seule
   if ( (out=fopen("sortie.txt","w"))==NULL)
       ERREUR("ouverture du fichier sortie.txt");

   //lecture des valeurs des nombres avec format %d
   fscanf(in,"%d, %d, %d, ",&a,&b, &nb);

   // écriture des valeurs des nombres en chaînes de caractères
   // avec format %d
   fprintf(out,"%d, %d, %d, ",a,b,nb);

   // équivalent printf, contrôle dans fenêtre console
   fprintf(stdout,"%d, %d, %d,\n",a,b,nb);

   // récupération des nb mots
   for (i=0; i<nb; i++){
       fscanf(in,"%s",buf);
       fprintf(out,"%s",buf);
       printf("%s ",buf);  // contrôle dans fenêtre console
   }
   return 0;
}
```

4. Sauvegarde d'éléments dynamiques

a. Sauver et récupérer un tableau dynamique

Lors d'une sauvegarde de données obtenues dynamiquement la taille peut être sauvegardée au début ou à la fin du fichier mais ce n'est pas obligatoire. En revanche lors du chargement des données il est nécessaire d'allouer la mémoire nécessaire des variables auxquelles sont affectées les données du fichier. Par exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef struct point{
    int x,y;
}t_point;

void    init        (t_point**t,int nb);
void    affiche     (t_point t[],int nb);

int main()
{
FILE*f;
t_point*data;
t_point*recup=NULL;
int nb;

   srand(time(NULL));
   if (f=fopen("test.bin","wb+")){
       // allocation et initialisation d'un tableau dynamique de
```

```
        // taille aléatoire
        nb=1+rand()%20;
        init(&data,nb);

        // affichege du tableau résultant
        affiche(data,nb);
        printf("-----------------------------\n");

        // enregistrement des nb structures du tableau sur le
        // fichier
        fwrite(data,sizeof(t_point),nb,f);

        // récupération dans un nouveau tableau dynamique alloué
        // séparément
        rewind(f);
        nb=0;
        while(!feof(f)){
            recup=(t_point*)realloc(recup,sizeof(t_point)*(nb+1));
            nb+=fread(&recup[nb],sizeof(t_point),1,f);
        }

        // affichage du résultat
        printf("load de %d struct t_point\n",nb);
        affiche(recup,nb);

        // libération de la mémoire allouée à la fin
        free(data);
        free(recup);
        // fermeture du fichier
        fclose(f);
    }
    else
        printf("erreur création fichier binaire\n");
    return 0;
}

void init(t_point**t, int nb)
{
int i;
    *t=(t_point*)malloc(sizeof(t_point)*nb);
    for (i=0; i<nb; i++){
        (*t)[i].x=rand()%100;
        (*t)[i].y=rand()%100;
    }
}

void affiche(t_point t[],int nb)
{
int i;
    for (i=0; i<nb; i++)
        printf("t[%3d].x=%3d,.y=%3d\n",i,t[i].x,t[i].y);
}
```

b. Récupérer des données via des pointeurs

Toute les allocations dynamiques de mémoire sont perdues lorsque le programme quitte. Lorsque des sauvegardes de données dynamiques sont faites sur fichier il faut toujours réallouer de la mémoire au moment de récupérer ces données dans le programme via des variables pointeurs. Par exemple :

```
int main()
{
FILE*f;
t_point data={10,20};
t_point*recup;

   if (f=fopen("test.bin","wb+")){
       printf("x=%d, y=%d\n",data.x, data.y);

       fwrite(&data,sizeof(t_point),1,f);
       rewind(f);

       recup=(t_point*)malloc(sizeof(t_point));
       fread(recup,sizeof(t_point),1,f);

       printf("x=%d, y=%d\n",recup->x, recup->y);
       free(recup);
       fclose(f);
   }
   else
       printf("erreur création fichier binaire\n");
   return 0;
}
```

5. Mise en pratique : Fichiers

Exercice 1
Dans un programme ouvrir en lecture un fichier dont le nom est donné par l'utilisateur. Indiquer si l'opération a réussi. Si l'opération échoue, le fichier n'existe pas, le créer et indiquer le résultat. Faire tourner le programme une première fois, le fermer et le lancer une deuxième fois pour vérifier que le fichier a bien été créé et s'ouvre correctement. Allez sur le disque voir ou le fichier se trouve. Ne pas oublier dans le programme de fermer le fichier quand il n'y a plus besoin de l'avoir ouvert.

Exercice 2
Faire un programme qui ouvre ou crée un fichier dont le nom, l'emplacement et le mode sont entrés par l'utilisateur. Ne pas oublier de fermer le fichier lorsque ça a réussi et qu'un message a été transmis à l'utilisateur.

Exercice 3
Un programme crée un fichier qui se trouve dans un dossier nommé test dans le répertoire du programme. Vérifier que le fichier est bien sur le disque dur. Modifier le programme pour ouvrir le fichier en lecture uniquement et relancer le programme. ne pas oublier de fermer le fichier.

Exercice 4
Avec le bloc note créez un fichier que vous sauvez dans un répertoire distant du programme.
Par rapport au programme il faut remonter de trois niveaux et redescendre de quatre niveaux pour trouver le fichier. Faire le programme qui ouvre le fichier et le referme.

Exercice 5
Ecrire un programme qui demande à l’usager d’entrer des mots un à un, qui stocke ces mots à la fin d’un fichier texte, puis qui affiche le contenu du fichier complet.

Initialement, le programme doit créer le fichier s'il n'existe pas, mais ne doit pas perdre son contenu s'il existe déjà. Vérifier ce point en lançant deux fois de suite le programme.

Exercice 6

Écrire un programme qui modifie le contenu d'un fichier texte comme suit :
Initialement le fichier texte contient sur chaque ligne une opération arithmétique sur deux entiers par exemple :
3 + 5
4 – 7
2 * 9
Le programme doit interpréter correctement le contenu de chaque ligne et écrire le résultat de l'opération dans le même fichier à la suite de chaque opération :
3 + 5 = 8
4 - 7 = -3
2 * 9 = 18
Se limiter aux quatre opérations arithmétiques de base (+, -, *, /) mais gérer correctement les erreurs courantes :

- fichier inaccessible ou inexistant
- format de fichier incorrect
- opération arithmétique inconnue.

Exercice 7

Ecrire un programme qui à partir d'un fichier texte détermine :

- le nombre de caractères qu'il contient
- le nombre de chacune des lettres de l'alphabet (on ne considérera que le minuscules)
- le nombre de mots
- le nombre de lignes

Les fins de ligne ne sont pas comptabilisées dans les caractères. On admettra que deux mots sont toujours séparés par un ou plusieurs des caractères suivants :

- fin de ligne
- espace
- ponctuation (, ; : ! . ?)
- parenthèses ( )
- guillemets "
- apostrophe '

On admettra également, pour simplifier, qu'aucun mot ne peut être commencé sur une ligne et finir sur la suivante.

Il est conseillé de réaliser une fonction permettant de décider si un caractère donné, transmis en argument, est un des séparateurs mentionnés ci-dessus. Elle renvoie 1 si oui et 0 si non.

Exercice 8

Soit un fichier texte créé avec le bloc note. Écrire un programme qui peut exécuter les opérations suivantes :

- sauvegarder le contenu du fichier texte en fichier binaire
- récupérer les données sur le fichier binaire
- modifier les données.

Chacune de ces opérations est contrôlée par un affichage dans la fenêtre console (au départ contenu du fichier texte, récupération sur fichier binaire, chaque nouvelle modification effectuée sur les chaines, chaque nouvelle récupération du fichier binaire).

Exercice 9
Soit une matrice dynamique de nombres dans un programme de nb1*nb2 aléatoires, faire les fonctions suivantes :

- □ une fonction d'initialisation avec des valeurs aléatoires
- □ une fonction d'affichage
- □ deux fonctions, une save et une load, de la matrice entière
- □ deux fonctions, une save et une load, de la matrice ligne par ligne
- □ deux fonctions, une save et une load, de la matrice nombre par nombre

Tester dans un programme qui donne le choix entre ces opérations via un menu utilisateur. Commencez par initialisation et affichage, puis ajoutez saves et loads, tester à chaque étape.

Exercice 10
Créez un programme permettant de saisir un mot au clavier puis de le stocker dans un fichier avant de le relire dans le fichier pour l'afficher à nouveau. Dans cet exercice, le fichier sera ouvert au début du main() et ne sera refermé qu'à la fin du main(), pas d'ouvertures puis de refermetures multiples.

Exercice 11
Dans un programme écrire une phrase avec plusieurs mots dans un fichier puis afficher le contenu du fichier à l'envers, lettre par lettre, en commençant par la fin.

Exercice 12
Dans un programme, faire une fonction qui lit les caractères dans un fichier texte selon un pas spécifié en paramètre. Par exemple 2 pour une lettre sur deux, 3 pour une lettre sur trois, 10 pour une lettre sur dix etc. Lorsque le pas est positif on avance du début vers la fin, Si le pas est négatif on va de la fin vers le début.

Exercice 13
Faire une fonction de lecture de caractères qui peut afficher n'importe quel caractère d'un fichier texte à partir de la position courante dans le fichier. Le déplacement en positif ou négatif dans le fichier est spécifié en paramètre de cette fonction d'affichage.

Exercice 14
Un magasin d'articles de sport a une base de donnée pour gérer son fond. Pour chaque article il faut stocker en mémoire : le code (valeur entière), le nom du produit, le prix unitaire du produit, le stock disponible (valeur entière).
La base de données est un fichier binaire. Toutes les informations des articles sont stockées article par article, les uns à la suite des autres sur le fichier.

Dans un programme, proposer à l'utilisateur les actions suivantes :

- □ 1) Définir un type approprié pour identifier les articles sachant que toutes les opérations se font en accès directe sans stockage de la base en mémoire centrale (RAM)
- □ 2) Ecrire une fonction de saisie d'un nouvel article
- □ 3) Ecrire une fonction de sauvegarde d'un article dans la base

- □ 4) Ecrire une fonction d'affichage d'un article dont le code est passé en paramètre
- □ 5) Faire une fonction d'affichage de la base.
- □ 6) Ecrire une fonction de recherche du code d'un article à partir de sa dénomination
- □ 7) Ecrire une fonction permettant à un utilisateur de modifier un article dont le code est passé en paramètre
- □ 8) Ecrire une fonction de suppression d'un article de la base soit par son code soit par son nom.

Exercice 15

Reprendre l'exercice 14 mais cette fois avec possibilité de chargement de la base dans le programme,

1) ajouter :
   - □ une fonction de chargement de la base
   - □ une fonction de sauvegarde de la base
2) modifier les autres fonctions en conséquence.