Programmation

En Langage C

Chaps 3 - 4 - 5

- Les Tableaux, Les structures,
- Les pointeurs et
- Les fonctions

Cours TD TP

Sommaire

Chap III : Les tableaux et les structures de données

- Les Tableaux : Déclaration, Accès
- Les Enregistrements : Déclaration, Accès

Chap IV : Les Pointeurs et allocation dynamique de mémoire

- Définition d'un pointeur
- Déclaration d'un pointeur
- Règles d'utilisation des pointeurs
- Arithmétique des pointeurs
- Pointeurs sur des structures
- Fonction d'allocation de mémoire

Sommaire

Chap V : Les fonctions

- Déclaration d'une fonction
- Définitions d'une fonction
- appel d'une fonction
- Exemple de fonctions standards
- Tableau transmis en argument
- Pointeur sur des fonctions
- Fonctions récursives

Les tableaux

Type tableau

Un type tableau est une représentation abstraite d'un ensemble de données de **même type**.

Un type tableau est créé par :

```
typedef    nom_type_elt    nom_type_Tableau[Taille1][Taille2][…] ;
```

Exemples

- Pour générer les types tableaux nommés **tvecteur** et **tmatrice** de 4 réels et de 3x3 réels :

```
typedef    float    tvecteur[4] ;
typedef    float    tmatrice[3][3] ;
```

- Pour générer les types tableaux nommés **tchaine et tlisteTexte** qui représentent respectivement les textes de longueur 30 caractères et les listes de 10 textes de 30 caractères chacun :

```
typedef    char  tchaine[30] ;
typedef    tchaine  tlisteTexte[10] ;
```

Remarque

- On peut construire des types de dimension supérieure à 2, mais pratiquement on se limite à la dimension 2.

Chap 3 : Les tableaux et les structures

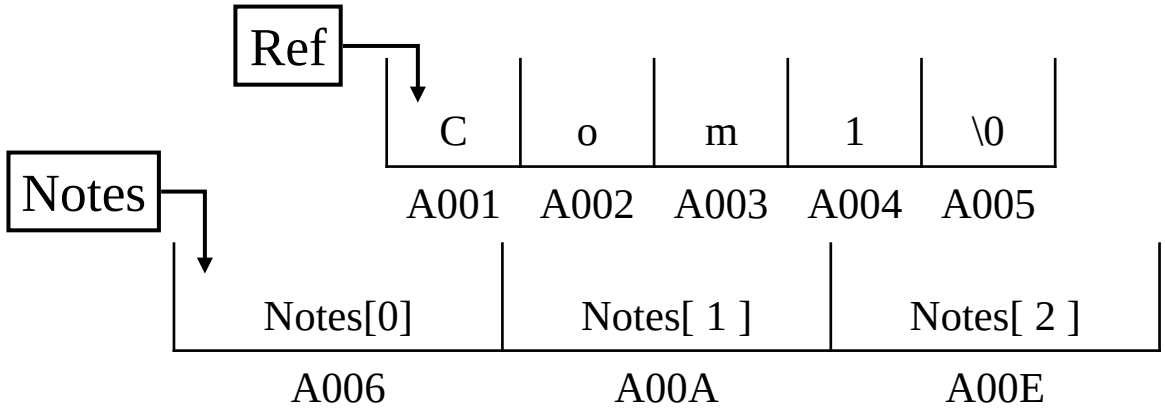
Déclaration de variables tableaux

Vecteur

```
nom_type _elt  nom_var_vect[Taille]={V1,V2,…};
nom_type_vect   nom_var_vect={V1,V2,…};
```

Exemples

```
   char    Ref[5]    ="Com1";
   float   Notes[3] ;
Ou bien
   typedef char  tchaine[5];
   typedef float  tnotes[3];
   tchaine        Ref="Com1" ;
   tnotes         Notes;
```



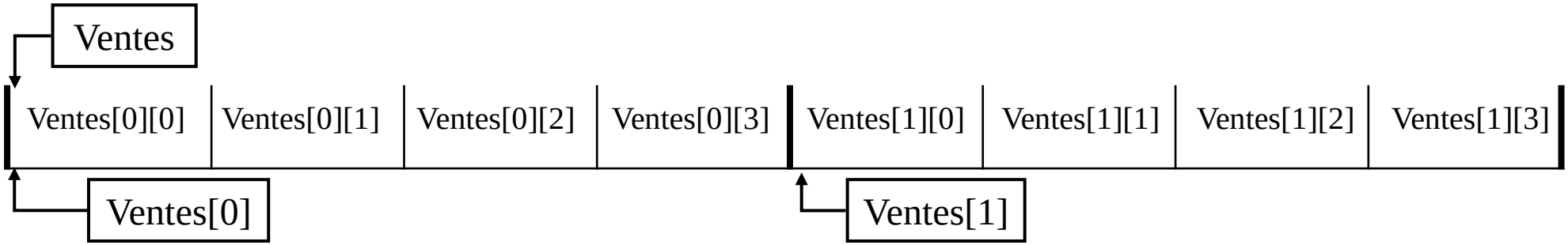
Matrice

Une variable matrice peut être déclarée et initialisée par :

```
nom_type_elt       nom_var_mat[Taille1][Talle2]={{V11,V12,…},{…},…..,{…}};
Ou bien :
nom_type_mat       nom_var_mat={{V11,V12,…},{…},…..,{…}};
```

Exemple Tableau des ventes trimestrielles pour les 2 dernières années

```
float Ventes [2][4] ;
ou bien
typedef float tventes[2][4];
tventes Ventes;
```



Remarque : Le nom d'une variable tableau désigne toujours une **adresse constante** : celle du premier élément du tableau. Par conséquent si T1 et T2 sont deux variables tableaux , l'écriture suivante **T1= T2** n'est pas possible et engendre une erreur.

Déclaration d'un tableau de Constantes

```
const type nom_const[taille] = {val1,Val2,…} ,… ;
```

Exemple *const float Ux[3] = {1,0,0}, Uy[ ]={0,1,0}, Uz[ ]={0,0,1}* ;

Accès aux éléments d'un tableau

Cas d'un vecteur

Pour un tableau à une seule dimension, les éléments sont indicés à partir de zéro. Ainsi l'accès à l'élément d'indice i se fait par :

```
Nom_Var_Tab[i]
```

Pour saisir une donnée dans cet élément

```
scanf("…" ,&Nom_Var_Tab[i]) ;
```

Pour affecter à cet élément, on écrit

```
Nom_Var_Tab[i] = Expression ;
```

Pour afficher le contenu de cet élément

```
printf("…" ,Nom_Var_Tab[i]) ;
```

Cas d'une matrice

Pour un tableau à deux dimensions, les lignes et les colonnes sont indicées à partir de zéro. Ainsi l'accès à l'élément de la ligne i et de la colonne j se fait par :

```
Nom_Var_Tab[i][j]
```

Pour affecter à cet élément, on écrit

```
Nom_Var_Tab[i][j] = Expression ;
```

Pour afficher le contenu de cet élément

```
printf("…" ,Nom_Var_Tab[i][j]) ;
```

Pour saisir une donnée dans cet élément

```
scanf("…" ,&Nom_Var_Tab[i][j]) ;
```

II. Les Structures

Type structuré (enregistrement)

Un type structuré est une définition abstraite d'un ensemble de données hétérogènes (ne sont pas tous de même type) caractéristiques d'une entité réelle (objet).

Un type structuré est obtenu par :

```
typedef   struct  nom_struct  {
            type_1   nomChamp_1   ;
            …
            type_n      nomChamp_n   ;
        } Nom_type_structure ;
```

Remarque : Le type d'un champ de la structure peut être simple ou à son tour un type construit.

Exemples

Le type Complexe composé de la partie réelle et la partie imaginaire :	Le type Employé caractérisé par un matricule, un nom, une date d'embauche et un salaire :

`typedef  struct  Tcomplexe {`
`  float  Preel ;`
`  float Pimag;`
`} tcomplexe  ;`
	`typedef  struct {`
	`  int  Matricule  ;`
	`  char Nom[20];`
	`  struct {int jour,mois,annee;}DateEmb;`
	`  float  Salaire  ;`
	`} temploye ;`

Exemple de type tableau de structure

Le type liste d'étudiant qui représente une classe de 20 étudiants :

```
typedef   struct {
    int     Matricule   ;
    char Nom[20];
    } tetudiant  ;
typedef  tetudiant  tliste_etudiant[20];
```

Exemple de type structuré composé de tableau

Le type Facture qui représente en plus des données propres à une facture, une liste de produits.
On suppose que le nombre de produits par facture est limité à 20.

```
typedef   struct {
    int     jour,mois,annee ;
} tdate   ;
typedef   struct {
    int Reference ;
    char Designation[20];
    float  Prix;
}tproduit    ;
typedef  tproduit  tliste_produit[20];
```

```
typedef   struct {
    int CodeClient    ;
    char NomClient[20];
    char  AdresClient[50];} tclient   ;
typedef   struct {
    int NumFacture ;
    tdate DateFacture  ;
    tclient clt;
    tliste_produit  Articles ;} tfacture ;
```

Exercices

Ecrire les modèles de structures qui décrivent une bibliothèque qui est caractérisée par :

- Un nom , Une adresse, Un tél , Un émail
- Une liste de livres pouvant contenir un maximum de 1000 livres.

Chaque livre est défini par :

- Un titre, Une référence, Un auteur, Un éditeur, Une date d'édition, un nombre de pages et une langue.

On aimerai définir chaque auteur par son nom , son sexe et sa nationalité.

Réponse

```
typedef   struct {
    int     jour,mois,annee ;
} tdate   ;
typedef   struct {
    char nom[30],sexe[2],nationalite[30];
}tauteur ;
typedef struct {
      char ref[10],titre[100],resume[1000]
      ,editeur[100],langue[50];
      int  nbrepage;
      tdate dateedition;
      Tauteur auteur;
}tlivre;
```

```
typedef   struct {
     char nom[100], adresse[200], tel[15];
     char  email[100];
     tlivre livres[1000];
   } tbiblio   ;
```

Chap 3 : Les tableaux et les structures

Variables structurée Une variable structurée peut être déclarée de plusieurs façons :

Façon 1

```
struct  nom_struct  {
    type_1   nomChamp_1    ;
    …
    type_n       nomChamp_n    ;
} NomVar   1,NomVar2, … ;
```

Façon 2

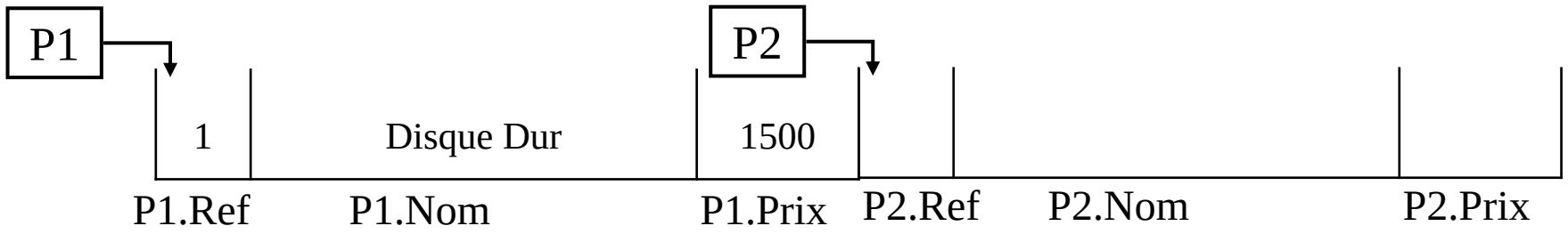
```
struct  nom_struct  {
    type_1   nomChamp_1    ;
    …
    type_n       nomChamp_n    ;
};
…
struct  nom_struct NomVar 1,NomVar2, …;
```

Façon 3

```
typedef struct  nom_struct  {
    type_1   nomChamp_1    ;
    …
    type_n       nomChamp_n    ;}Nom_type_structure;
...
Nom_type_structure  NomVar  1,NomVar2, …;
```

Une variable structurée peut être initialisée lors de sa déclaration en plaçant les valeurs des champs entre accolades comme est indiqué dans l'exemple suivant :

```
typedef struct {
    int     Ref;
    char Nom[12];
    float Prix;}tproduit;
tproduit  P1={1,"Disque Dur",1500}, P2;
```



Remarque : Si X et Y sont deux variables structurées de même type, l'écriture **X=Y** est possible et permet de copier champ par champ, les valeurs des champs de Y dans ceux de X.

Déclaration de Constantes structurée

```
const type_structure nom_const = {val1,Val2,…} ,… ;
```

Exemple *const tcmplexe Nul = {0,0}, ImgPur = {0,1}* ;

Accès aux champs d'une variable structurée

L'accès à u champs d'une variable structurée se fait par :

```
Nom_Var_Str.NomChamp
```

Pour affecter à cet élément, on écrit

```
Nom_Var_Str.NomChamp = Expression ;
```

Pour saisir une donnée dans cet élément

```
scanf("…" ,&Nom_Var_Str.NomChamp) ;
```

Pour afficher le contenu de cet élément

```
printf("…" ,Nom_Var_Str.NomChamp) ;
```

Application

Application sur le calcul des polynômes :

Somme de deux polynômes

```
/* Programme Somme de deux polynômes */

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

#define N 100

typedef struct {
      int d;
      double coeff[N];
}tpolynome;


int main(){
      //déclaration des polynomes
      tpolynome P,Q,S;
      int min, i;
      //Saisie de P
      puts("Saisie de P : ");
      printf("entre de degré : ");
      scanf("%d",&P.d);
      for(i=0;i<=P.d;i++){
            printf("Coeff[%d] : ",i);
            scanf("%lf",&P.coeff[i]);
      }

      //Saisie de Q
      puts("Saisie de Q : ");
      printf("entre de degré : ");
      scanf("%d",&Q.d);
      for(i=0;i<=Q.d;i++){
            printf("Coeff[%d] : ",i);
            scanf("%lf",&Q.coeff[i]);
      }
```

```
        //Calcul de la somme
        min=P.d<Q.d?P.d:Q.d;
        S.d=P.d>Q.d?P.d:Q.d;
        for(i=0;i<=min;i++)
                S.coeff[i]=P.coeff[i]+Q.coeff[i];
        for(;i<=P.d;i++)
                S.coeff[i]=P.coeff[i];
        for(;i<=Q.d;i++)
                S.coeff[i]=Q.coeff[i];

        //Affichage de S
        puts("Affichage du résultat S : ");
        printf("Le degré : %d\n",S.d);
        for(i=0;i<=S.d;i++)
                printf("Coeff[%d] : %f\n",i,S.coeff[i]);
        system("pause");
        return 0;
}
```

I. Pointeurs

1. Définition

Un pointeur est une variable dont le contenu est une adresse d'une case mémoire. Son contenu étant variable, un pointeur permet donc de pointer , dans le temps, sur différentes cases mémoires.

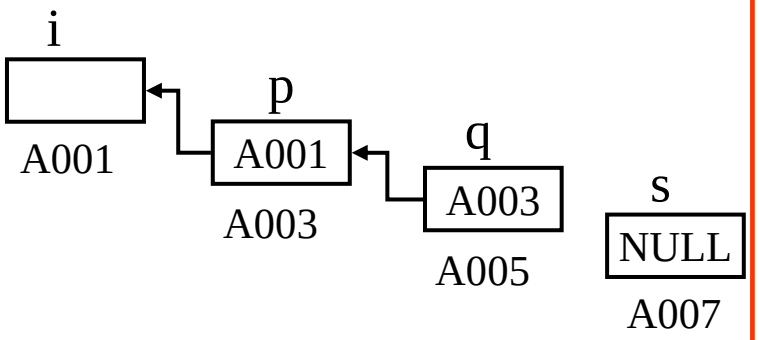
2. Déclaration d'un pointeur

La déclaration d'un pointeur, similaire à celle d'une variable, doit indiquer le nom du pointeur, le type de la case à pointer et éventuellement la valeur initiale du pointeur(adresse initiale) :

```
Nom_type  *nom_pointeur[=adresseInitiale];
```

Exemples

```
int  i, *p=&i, **q=&p; /*pointeur sur un entier*/
char *s=NULL; /* pointeur sur un caratère */
```



3. Remarques

- On doit toujours affecter une adresse à un pointeur avant de pouvoir le manipuler.
- Pour exprimer qu'un pointeur ne pointe nulle part, on peut l'initialiser à la valeur **NULL.**
- Le contenu d'un pointeur peut être défini de deux manières différentes :
  - En utilisant l'opérateur adresse de **&**
  - En utilisant les fonctions d'allocation dynamique de mémoire : **malloc**, **calloc**, **realloc**, …
- Le contenu de la case pointée par un pointeur peut être référencé en utilisant l'opérateur ***

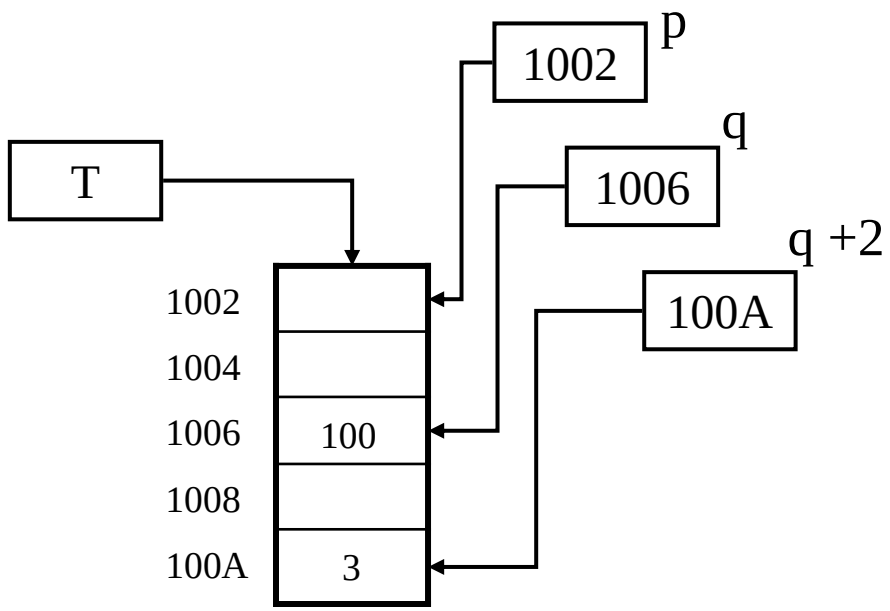
4. Arithmétique des pointeurs

La valeur d'un pointeur est entier, on peut lui appliquer un certain nombre d'opérations arithmétiques visant à déplacer le pointeur dans le plan mémoire d'un nombre de cases du type le type de la case pointée. Les opérations valides pour les pointeurs sont :

- L'**addition** d'un entier relatif à un pointeur: le résultat est un pointeur de même type que pointeur de départ.

Exemple

```
int   T[5], *p=T , *q ;
…
q=p+2 ;
*q=100 ;
*(q+2)=3 ;
```



- La **différence** de deux pointeurs pointant tous les deux sur des objets de même type appartenant à une zone contiguë (cas de deux éléments d'un même tableau). Le résultat est **l'entier relatif** qui représente en valeur absolue le **nombre de cases** séparant les deux pointeurs.

Exemple

```
int n  ;
n = q - p ;  /* n reçoit 2  */
N= p – q ; / n reçoit – 2 */
```

- **Incrémentation(décrémentation)** : l'opérateur ++(--) déplace un pointeur vers la case mémoire suivante (précédente)
- **Comparaison** de deux pointeurs de même type en utilisant les opérateurs de comparaison (= =, !=, <,<=,> et >=). Le résultat est logique (0 pour Faux et ≠ pour Vrai)

Exemple: Saisie des données d'un tableau d'entiers T en utilisant un pointeur p.

```
int   T[5], *p;
...
for(p=T; p < T + n ; p++ ){
    printf("T[%d]= ", p-T);
    scanf("%d", p);
}
```

Exemple : Manipulation d'un tableau à l'aide des pointeur : On cherche à :

- réserver un tableau de 100 réels. Le nombre de réels placés dans le tableau est noté n.
- Saisir les données du tableau,
- Rechercher les adresses du min et du max du tableau
- Afficher les données du tableau
- Afficher le min et le max du tableau et leurs positions dans le tableau.

5. Pointeur et tableau

Cas d'un vecteur

Un tableau est une vu comme un pointeur constant (pointeur sur le 1er élément). Il est alors possible d'utiliser le nom d'un tableau dans les écritures suivantes :

Pour affecter à cet élément, on écrit

```
Nom_Var_Tab[i] = Expression ;
Ou bien
*(Nom_Var_Tab +i)=Expression ;
```

Pour saisir une donnée dans cet élément

```
scanf("…" ,&Nom_Var_Tab[i]) ;
Ou bien
scanf("…", Nom_Var_Tab+i);
```

Pour afficher le contenu de cet élément

```
printf("…" ,Nom_Var_Tab[i]) ;
Ou bein
printf("…" ,*(Nom_Var_Tab+i)) ;
```

Cas d'une matrice

Pour affecter à cet élément, on écrit

```
Nom_Var_Tab[i][j] = Expression ;
Ou bien, si N est le nombre de Colonne:
*((type_Elt)Nom_Var_Tab + i*N + j ) = Expression ;
```

Pour saisir une donnée dans cet élément

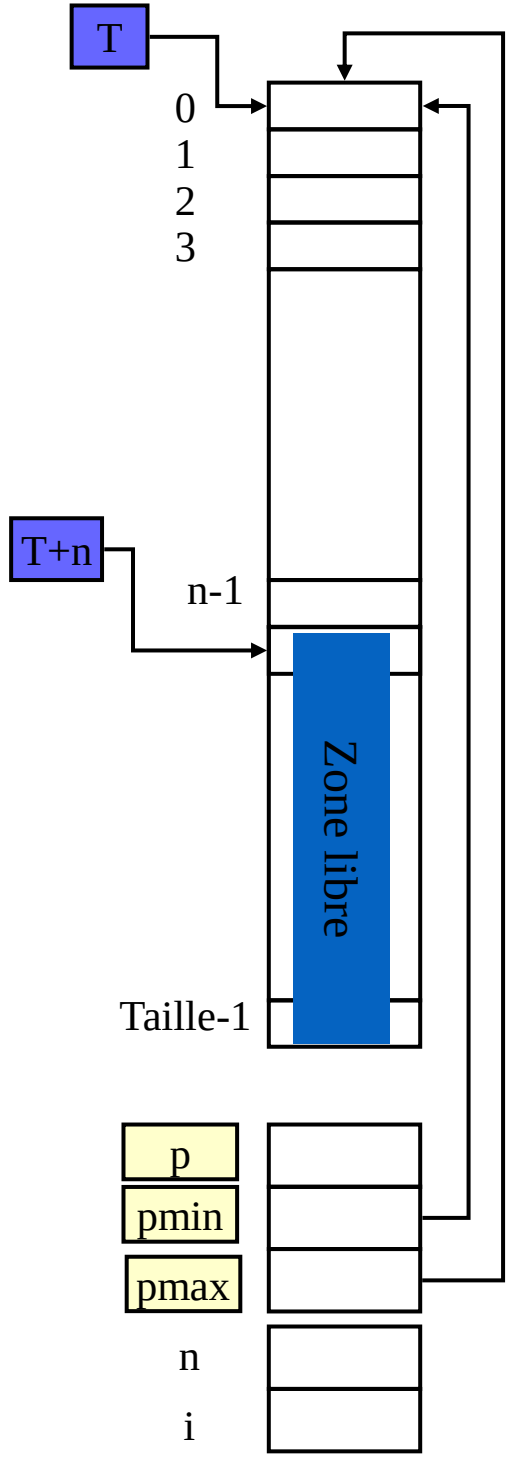
```
scanf("…" ,&Nom_Var_Tab[i][j]) ;
Ou bien, si N est le nombre de colonnes :
scanf("…", (Type_Elt)Nom_Var_Tab + i * N + j );
```

Pour afficher le contenu de cet élément

```
printf("…" ,Nom_Var_Tab[i][j]) ;
Ou bein, si N est le nombre de colonne :
printf("…" ,*((Type_Elt)Nom_Var_Tab+ i*N  + j )) ;
```

Chap 4 : Pointeurs et Allocation dynamique de mémoire

```
#include<stdio.h>
#include<conio.h>
#define taille 100;
float T[taille], *p, *pmin,*pmax;
Int n,i;
void main(){
      printf("taille effective du tableau : "); scanf("%d",&n) ;
      /* saisi du tableau */
      for(p =T ; p <T+n ; p++){
          printf("T[%d] = ",p-T); scanf("%f",p);}
          /*pointer sur le minimum et le maximum du tableau*/
      pmin=pmax=T;
      for( i =1 ; i<n;i++){
          if(T[i]<*pmin) pmin=T+i;
          if( T[i]>*pmax) pmax=T+i;}
      /*affichage du tableau */
      i=0;
      while(i<n){
          printf("T[%d]=%f\n",i,*(T+i));i++;}
/* affichage du min et du max */
      printf("le min = %f\tsa position est : %d\n",*pmin,pmin-T);
      printf("le max = %f\tsa position est : %d\n",*pmax,pmax-T);
      getch();
}
```



Adresse constante ☞ Un tableau est un pointeur constant
Adresse variable

6. Pointeur sur une variable structurée – Opérateur ->

L'accès au contenu d'un champs d'un bloc structuré pointé par un pointeur peut se faire à l'aide de l'opérateur d'indirection -> par :

```
Nom_pointeur → Nom_champ
```

Cette écriture simplifie avantageusement l'écriture suivante :

```
(* Nom_pointeur) . Nom_champ
```

Exemple : Saisie et affichage des données d'une structure à l'aide d'un pointeur :

```
#include<stdio.h>
#include<conio.h>
struct Tpersonne{
     char nom[20];int age;};
struct Tpersonne X,*p;
void main(){
     /* saisie */
     p=&X;
     printf("Nom : "); gets(p →  nom) ;
     printf("Age : "); scanf("%d",&p →  age)
     /* affichage */
     printf("Nom : %s\nAge : %d",p →   nom,p →   age);
     getch();
}
```

II. Allocation dynamique

1. Variable dynamique

Une variable dynamique est une variable qui peut être **créée** et **détruite** pendant **l'exécution**. La **taille** d'une telle variable peut être **modifiée** aussi au cours de l'exécution.

2. Création d'une variable dynamique

L'allocation dynamique d'un bloc mémoire se fait par l'appel à aux fonctions d'allocation **malloc** ou **calloc**. Ces fonctions retournent l'adresse du bloc réservé. Cette adresse doit être conservée dans une variable pointeur :

```
Type *nom_pointeur ;
..
Nom_pointeur=(type *) malloc(Taille_Bloc);
Nom_pointeur=(type *)calloc(Nbre_Elements, Taille_Elment);
```

Remarques

- La taille indiquée aux fonctions malloc et calloc doit être multiple de la taille du type des données à mémoriser. On utilise pour cela la fonction **sizeof** qui retourne la taille occupée par un type.
- Lorsque l'allocation dynamique ne peut pas se faire, les fonctions malloc et calloc retourne la valeur NULL.

3. Libération d'une variable dynamique

Un bloc mémoire réservé par malloc ou calloc et pointée par un pointeur p peut être détruit par l'appel à la fonction **free** :

```
free( p ) ;
```

Exemple

```
Int *p,n;
…
Scanf("%d",&n);
P=(int *)calloc(n,sizeof(int));
for(i=0;i<p+n;i++) scanf("%d",p+i);
…
free(p);
```

Remarques

- free ne peut être utilisée que pour libérer des blocs préalablement alloués par malloc ou calloc.
- L'affectation de la valeur NULL à un pointeur qui pointe sur un bloc dynamique risque d'entraîner un bloc inaccessible.

4. Réallocation de mémoire

La taille d'un bloc mémoire dynamique pointée par un pointeur p peut être augmentée ou diminuée à l'aide de la fonction **realloc** :

```
p = (type *)realloc( p, nouvelle_taille) ;
```

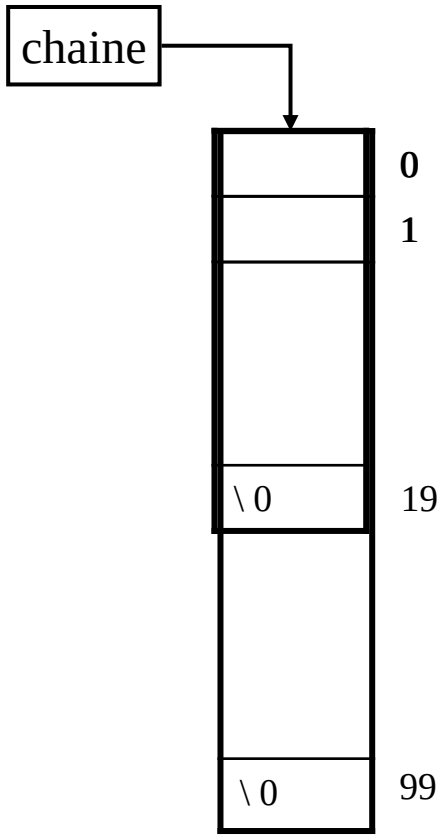
Si la réallocation n'est pas possible, realloc renvoie la valeur NULL.

Exemple 1 : Manipulation d'une chaîne de caractères dynamique avec redimensionnement.

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

char *chaine;

void main()
{
   clrscr();
   /* allocation  de 20 caratères*/
   chaine=(char *) malloc(20);
   gets(chaine);
   /* réallocation  à 100 caractères */
   chaine=(char *)realloc(chaine,100);
   puts(chaine);
   /* Concaténation */
   chaine=strcat(chaine," ceci est une chaîne constante ");
   puts(chaine);
   getch();
}
```



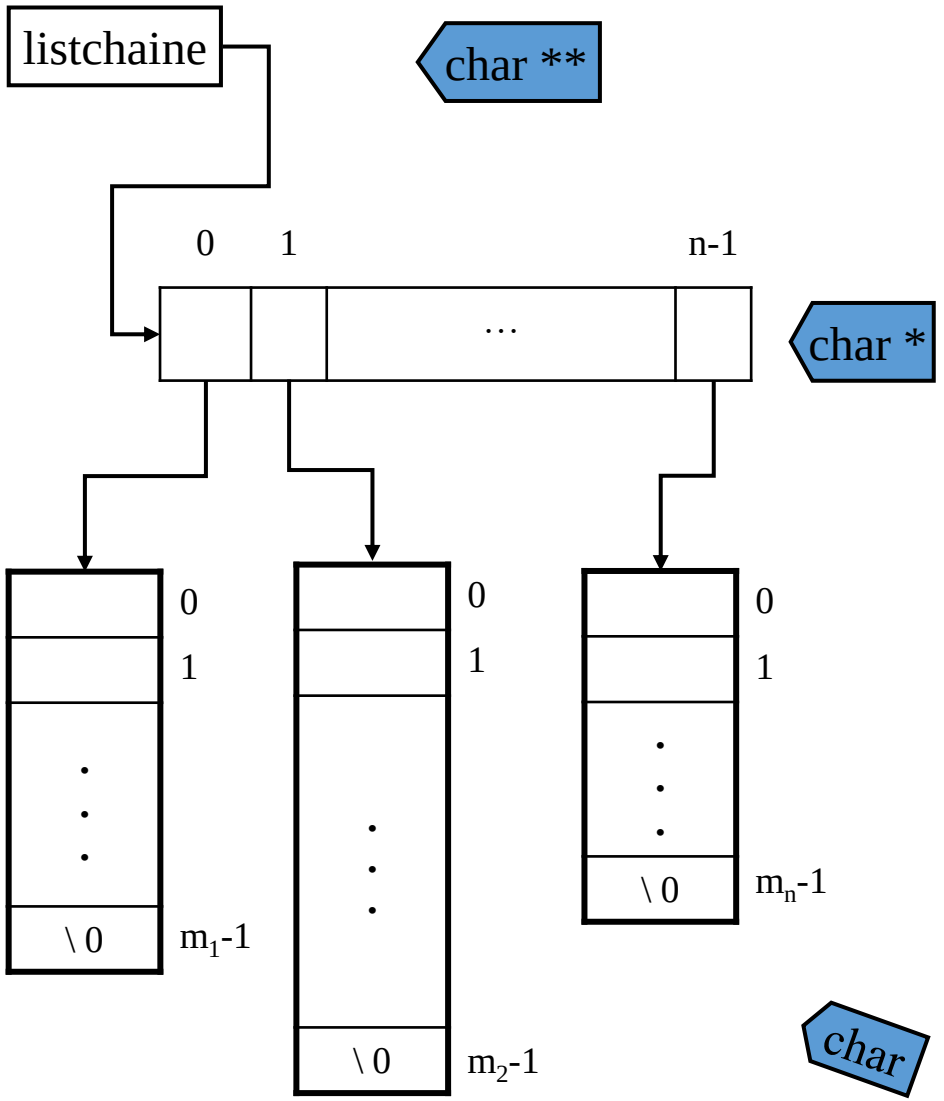
Exemple 2 : Manipulation d'une liste dynamique de chaînes de caractères dynamiques.

Chap 4 : Pointeurs et Allocation dynamique de mémoire

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

char **listchaine,**p;
int n,m;

void main()
{
   clrscr();
   /* allcation des pointeurs */
   printf("nbre de chaines : ");scanf("%d",&n);
   listchaine=(char **)malloc(n*sizeof(char *));
   /* allocation des chaines */
   for(p=listchaine;p<listchaine+n;p++)
   {
        printf("taille de la chaine %d :",p-listchaine+1);
        scanf("%d",&m);
        *p=(char *)malloc(m);
    }
   /* lecture des chaînes*/
   for(p=listchaine;p<listchaine+n;p++){
        fflush(stdin);
        printf("chaine %d :",p-listchaine+1);
        gets(*p);
   }
/* affichage des chaines */
   for(p=listchaine;p<listchaine+n;p++)
        puts(*p);
   getch( );
}
```



5. Memory Errors

- **Using memory that you have not initialized**
- **Using memory that you do not own**
- **Using more memory than you have allocated**
- **Using faulty heap memory management**

4.1. Using memory that you have not initialized

- Uninitialized memory read
- Uninitialized memory copy
  - not necessarily critical – unless a memory read follows

```
void f(int *pi) {
  int j;
  *pi = j;
  /* UMC: j is uninitialized, copied into *pi */
}
int main() {
  int i=10;
  f(&i);
  printf("i = %d\n", i);
  /* UMR: Using i, which is now junk value */
}
```

4.2. Using memory that you don't own

- Null pointer read/write
- Zero page read/write

```
typedef struct node {
  struct node* next;
  int val;
} Node;

int findLastNodeValue(Node* head) {
  while (head->next != NULL) { /* Expect NPR */
      head = head->next;
  }
  return head->val; /* Expect ZPR */
}
```

What if head is NULL?

4.2. Using memory that you don't own

- Invalid pointer read/write
  - Pointer to memory that hasn't been allocated to program

```
void genIPR() {
  int *ipr = (int *) malloc(4 * sizeof(int));
  int i, j;
  i = *(ipr - 1000); j = *(ipr + 1000); // Expect IPR
  free(ipr);
}

void genIPW() {
  int *ipw = (int *) malloc(5 * sizeof(int));
  *(ipw - 1000) = 0; *(ipw + 1000) = 0; // Expect IPW
  free(ipw);
}
```

4.2. Using memory that you don't own

- Common error in 64-bit applications:
  - `int`s are 4 bytes but pointers are 8 bytes
  - If prototype of `malloc()` not provided, return value will be cast to a 4-byte `int`

Four bytes will be lopped off this value – resulting in an invalid pointer value

```
/*Forgot to #include <malloc.h>, <stdlib.h>
  in a 64-bit application*/
void illegalPointer() {
  int *pi = (int*) malloc(4 * sizeof(int));
  pi[0] = 10; /* Expect IPW */
  printf("Array value = %d\n", pi[0]); /* Expect IPR */
}
```

4.2. Using memory that you don't own

- Free memory read/write
  - Access of memory that has been freed earlier

```
int* init_array(int *ptr, int new_size) {
  ptr = (int*) realloc(ptr, new_size*sizeof(int));
  memset(ptr, 0, new_size*sizeof(int));
  return ptr;
}
int* fill_fibonacci(int *fib, int size) {
  int i;
  /* oops, forgot: fib = */ init_array(fib, size);
  /* fib[0] = 0; */ fib[1] = 1;
  for (i=2; i<size; i++)
      fib[i] = fib[i-1] + fib[i-2];
  return fib;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?

4.2. Using memory that you don't own

- Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
  const int MAXSIZE = 128;
  char result[128];
  int i=0, j=0;
  for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
      result[i] = s1[j];
  }
  for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
      result[i] = s2[j];
  }
  result[++i] = '\0';
  return result;
}
```

`result` is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

4.3. Using memory that you haven't allocated

- Array bound read/write

```
void genABRandABW() {
  const char *name = "Safety Critical";
  char *str = (char*) malloc(10);
  strncpy(str, name, 10);
  str[11] = '\0'; /* Expect ABW */
  printf("%s\n", str); /* Expect ABR */
}
```

4.4. Using faulty heap memory management

- Memory leak

```
int *pi;
void f() {
  pi = (int*) malloc(8*sizeof(int));
  /* Allocate memory for pi */
  /* Oops, leaked the old memory pointed to by pi */
  …
  free(pi); /* f() is done with pi, so free it */
}

void main() {
  pi = (int*) malloc(4*sizeof(int));
  /* Expect MLK: f() leaks it */
  f();
}
```

4.4. Using faulty heap memory management

- Potential memory leak
  - no pointer to the beginning of a block
  - not necessarily critical – block beginning may still be reachable via pointer arithmetic

```
int *plk = NULL;
void genPLK() {
  plk = (int *) malloc(2 * sizeof(int));
   /* Expect PLK as pointer variable is incremented
      past beginning of block */
  plk++;
}
```

4.4. Using faulty heap memory management

- Freeing non-heap memory
- Freeing unallocated memory

```
void genFNH() {
  int fnh = 0;
  free(&fnh); /* Expect FNH: freeing stack memory
  */
}
void genFUM() {
  int *fum = (int *) malloc(4 * sizeof(int));
  free(fum+1); /* Expect FUM: fum+1 points to
  middle of a block */
  free(fum);
  free(fum); /* Expect FUM: freeing already freed
  memory */
}
```

Chap 5 : Les fonctions

1. Introduction

Une fonction représente le moyen de structurer un programme sous une forme modulaire.

Ceci présente les avantages suivants :

- Meilleure lisibilité
- Meilleur maintenance
- Possibilité de réutilisation

Une fonction est caractérisée par :

- son **prototype** : ligne en-tête qui indique le **nom** de la fonction, son **type** et ses **arguments**
- Son **corps** : ensemble de déclarations d'objets locaux (types, constantes, variables) à la fonction et une suite d'actions traduisant la tâche associée à la fonction.

2. Déclaration d'une fonction

```
type Nom_fonction (type1, type2, …) ;
```

Remarques

- Pour déclarer une procédure, il suffit de prendre **void** comme type de la fonction
- Les noms des arguments peuvent ne pas être mentionnés dans une déclaration
- Une fonction peut être sans argument
- Si le type de la fonction est omis, celui-ci sera le type **int** par défaut.

3. Définition d'une fonction

Une fonction C est définie selon la syntaxe suivante :

```
type Nom_fonction (type1 arg1, type2 arg2, …)
{
   /*déclarations locales */
   …
   /* Instructions */
   …
}
```

Remarques

- Les arguments d'une fonctions constituent des **objets locaux** à la fonction avec l'exception d'être initialisés par l'extérieur lors de l'appel.
- Le seul mode de communication en C est le **mode par valeur**
- Une communication **par adresse** peut être réalisée en utilisant un **pointeur** comme argument. Ceci permet à une fonction de modifier la valeur du paramètre effectif dont l'adresse lui a été communiquée.
- Si le type de la fonction est différent de void, le fonction retourne sa valeur à l'extérieur à l'aide de l'instruction **return** . L'instruction return entraine l'arrêt de la fonction.

return expression

- La transmission d'un tableau se fait toujours par adresse. En effet un tableau constitue l'adresse de son premier élément.
- Une fonction peut retourner une donnée de type simple, une donnée structurée ou une donnée de type pointeur.

4. Appel à une fonction

L'appel à une fonction se fait par

```
Nom_fonction (argeff1, argeff2, …) ;
```

Remarques

- Les arguments effectifs et arguments formels doivent se correspondre en nombre et en type.
- La déclaration d'une fonction est nécessaire si l'appel à la fonction se fait avant sa définition.

Exemples

Exemple1 : Fonction factorielle

```
long fact (int n) {
    long f =1;
    int i;
    For(i=1;i<=n;i++) f=f*i;
    return f;
}
```

Exemple2 : Permutation de deux réels

```
void permuter (float *x , float *y) {
    float aux ;
    aux=*x;
    *x=*y;
    *y=aux;
}
```

Exemple 3 : Calcul du **salaire** des commerciaux d'une entreprise, composé d'un **fixe** et d'une **prime** de 10% du **chiffre d'affaire** réalisé si celui-ci dépasse les 50 000 DH, de 3% sinon. On isolera la suite d'actions qui permet de rentrer le salaire fixe ainsi que le chiffre d'affaire.

On impose les contraintes suivantes : Salaire Fixe ≥2000 et chiffre d'affaire ≥ 0

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

void Saisie(float *, float *) ;
float Prime(float ) ;

float  SalF, ChA, Salaire;

int  main(){
   Saisie(&SalF , &ChA);
   Salaire = SalF + Prime(ChA);
   printf("le salaire est : %.2f ", Salaire) ;
   system("pause");
   return 0;
}

void Saisie (float  *SF, float *CA){
   const float  Sal_Min = 2000. ;
   do{   printf("Entrer le salaire fixe : ") ;
         scanf("%f",SF);
   }while(*SF< Sal_Min);
   do{
      printf("Entrer le chiffre d'affaire : ") ;
      scanf("%f",CA);
   }while(*CA< 0);
}

float Prime (float CA )
{
   const float Plafond = 50000 ;
   float Taux ;
   if(CA >= Plafond)
      Taux = 0.1 ;
   else
      Taux = 0.03 ;
   return CA * Taux ;
}
```

5. Tableaux transmis en argument

Lorsque on place le nom d'un tableau en argument effectif d'une fonction, on transmet l'adresse du tableau à la fonction, ce qui lui permet de manipuler les éléments en lecture et en écriture.

Transmission d'un vecteur

Un argument vecteur d'une fonction peut être indiqué par :

```
Type_fonction nom_fonction (type_element T[ ],…)
```

Ou par :

```
Type_fonction nom_fonction (type_element  * T,…)
```

Transmission d'une matrice

Un argument matrice d'une fonction peut être indiqué en précisant le nombre de colonnes :

```
Type_fonction nom_fonction (type_element T[ ] [nbre_col] ,…)
```

Ou bien :

```
Type_fonction nom_fonction (type_element  * T,…)
```

Cette dernière façon exige un calcul de l'adresse de l'élément à manipuler. On accède à **T[i][j]** par ***(T+nbre_col*i +j)**

Remarque : Pour définir une fonction capable de recevoir un tableau de taille quelconque, il suffit de communiquer en plus de son adresse sa ou ses tailles en argument :

```
Type_fonction nom_fonction (type_element T[ ], int taille, …)
Type_fonction nom_fonction (type_element T[ ][ ], int taille1, int taille2, …)
```

Exemples

Exemple 1 : Saisie, Tri par ordre croissant en utilisant la méthode de tri par **sélection** et affichage d'un vecteur de n réels.

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

void saisir(float t[ ],int n);
void trier(float [ ],int);
void afficher(float [ ],int);
void permuter(float *, float *);

void main(){
    const nmax=5;
   int n;float v[nmax];
   do{   printf("nombred'éléments:");
         scanf("%d",&n);
   }while(n>nmax||n<=0);
   saisir(v,n);
   trier(v,n);
   afficher(v,n);
}

void afficher(float t[ ],int n){
   for(int i=0;i<n;i++)
       printf("t[%d]=%.2f\n",i,t[i]);}
```

```
void saisir(float t[ ],int n){
   for(int i=0;i<n;i++){
       printf("t[%d]=",i);
       scanf("%f",&t[i]);}
}

void permuter(float *x,float *y){
   float aux;
   aux=*x;
   *x=*y;
   *y=aux;
}

void trier (float t[ ],int n) {
   int i,j,p;
   for(i=0;i<n-1;i++){
       p=i;
       for(j=i+1;j<n;j++)
           if(t[j]<t[p])p=j;
       if(p!=i)permuter(&t[i],&t[p]);}
}
```

Exemple 2 : Saisie et affichage d'une matrice carrée 3x3

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

#define nmax 3

void saisie(float [ ][nmax]);
void afficher(float [ ][nmax]);

void main(){
   clrscr();
   float v[nmax][nmax];
   saisie(v);
   afficher(v);
   getch();
}

void saisie(float T[][nmax]){
   for(int i=0;i<nmax;i++)
       for(int j=0;j<nmax;j++){
           printf("T[%d][%d]=",i,j);
           scanf("%f",&T[i][j]);
       }
}
```

```
void afficher(float T[ ][nmax])
{
   for(int i=0;i<nmax;i++)
   {
       for(int j=0;j<nmax;j++)
           printf("%10.2f",T[i][j]);
       printf("\n");
   }
}
```

Exemple 2 bis : Saisie et affichage d'une matrice carrée n*n où n est variable

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

void saisie(float**,int);
void afficher(float** , int);
float ** creation(int);

void main(){
   float **v;int n; scanf("%d",&n);
   v=creation(n);
   saisie(v);
   afficher(v);
   getch();
}

void saisie(float **T,int n){
   for(int i=0;i<n;i++)
       for(int j=0;j<n;j++){
           printf("T[%d][%d]=",i,j);
           scanf("%f",&T[i][j]);
       }
}
```

```
void afficher(float **T,int n)
{
   for(int i=0;i<n;i++)
   {
       for(int j=0;j<n;j++)
           printf("%10.2f",T[i][j]);
       printf("\n");
   }
}

float **Creation(int n){
   float **m;
   m=(float **)malloc(n*sizeof(float*);
   for(int i=0;i<n;i++)
       m[i]=(float *)malloc(n*sizeof(float));
   return m;
}
```

Exemple 2 : Saisie et affichage de matrices carrées dont les tailles sont différentes →Approche pointeur :

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

void saisie(float *,int ,int );
void afficher(float *,int,int);

void main()
{
   float v[2][3], w[3][2] ;
   clrscr();
   saisie((float *)v,2,3);
   afficher((float *)v,2,3);
   getch();
   clrscr();
   saisie((float *)w,3,2);
   afficher((float *)w,3,2);
   getch();
}
```

```
void saisie(float *T,int n,int m)
{
   for(int i=0;i<n;i++)
       for(int j=0;j<m;j++)
       {
           printf("T[%d][%d]=",i,j);
           scanf("%f",T+m*i+j);
       }
}

void afficher(float *T,int n,int m)
{
   for(int i=0;i<n;i++){
       for(int j=0;j<m;j++)
           printf("%10.2f",*(T+i*m+j));
       printf("\n");
   }
}
```

6. Pointeurs sur des fonctions

En C, le nom d'une fonction pris seul désigne l'adresse de la fonction. Il est alors possible en affectant cette adresse à un variable pointeur de référencer la fonction..

6.1. Déclaration d'un pointeur sur une fonction

Un pointeur sur une fonction peut être défini par :

```
Type_fonction (* nom_pointeur)(type1 arg1,…)
```

Exemple :

```
float ( * pf ) (float , int ) ;
```

Dans cet exemple, *pf est une fonction à deux arguments (un réel et un entier) et à un résultat réel. pf désigne un pointeur sur une fonction retournant un réel et recevant un réel et un entier.

6.2. Déclaration d'un pointeur sur une fonction

L'appel à la fonction pointée par pf se fait par :

```
( * pf ) ( argeff1 , argeff2 ) ;
```

6.3. Application :

Programme C permettant de tracer les courbes représentatives des fonctions suivantes :

Cosinus

Sinus

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>

void tracer(float (*)(float));
float sinus(float);
float cosinus(float);

void main(){
   tracer(sinus);
   tracer(cosinus);}

void tracer(float (*pf)(float)){

   int a=DETECT,b,yemax,xemax;
   int xe,ye,i,npt;
   float pas,xmin,xmax,ymin,ymax,kx1,kx2,ky1
   ,ky2,x,y;

   puts("xmin : ");
   scanf("%f",&xmin);
   puts("xmax : ");
   scanf("%f",&xmax);
   puts("nombre de points : ");
   scanf("%d",&npt);

   ymax=1;
   ymin=-ymax;
   initgraph(&a,&b,"c:\\tc\\bgi");
   setcolor(1);
   setbkcolor(WHITE);
   xemax=getmaxx();
   yemax=getmaxy();

   pas=(xmax-xmin)/(npt-1);
   kx1=(xemax-240)/(xmax-xmin);
   kx2=(120*xmax-xmin*(xemax-120))/(xmax-xmin);

   ky1=(yemax-80)/(ymin-ymax);
   ky2=40-ky1*ymax;
   line(120,ky2,xemax-120,ky2);
   rectangle(120,40,xemax-120,yemax-40);
   xe=120;ye=40;
   for(i=0;i<npt;i++){
       moveto(xe,ye);
       x=pas*i+xmin;
       xe= x*kx1+kx2;
       y=(*pf)(x);
       ye=ky1*y+ky2;
       lineto(xe,ye);}
   getch();
   closegraph();}

float sinus(float x){ return sin(x);}

float cosinus(float x){ return cos(x);}
```

7. Fonctions récursives

7.1. Définitions

La **RECURSIVITE** est **une démarche** qui consiste à **faire référence** à ce qui fait **l'objet de la démarche** :

- c'est le fait de décrire un processus dépendant de données en faisant appel à ce même processus sur d'autres données plus «simples»,
- de montrer une image contenant des images similaires,
- de définir un concept en invoquant le même concept.

La **RECURSIVITE** Désigne pour un programme le fait de **s'auto-désigner**, ou de **s'auto-Invoquer.**

Une fonction qui contient un ou plusieurs **appels à elle-même** est dite **récursive**.

7.2. Règles à respecter :

1. Prévoir une **Condition d'arrêt** sinon appels infinis
2. Vérifier la progression vers la condition d'arrêt = Finitude
3. Ne jamais dédoubler du travail dans deux appels (éviter des calculs redondants)

7.3. Avantages et Inconvénients

Avantages :

- Formulation compacte, claire et élégante.
- Maîtrise des problèmes dont la nature même est récursive(structures de données hiérarchiques)

Inconvénients :

- Possibilité de grande occupation de la mémoire.
- Temps d'exécution peut être plus long.
- Estimation difficile de la profondeur maximale de la récursivité.
- Difficulté de la mise en place de la condition

7.4. Exemples :

Factorielle d'un nombre :

Fact(n) = Fact(n-1)*n

Fact(0)=1

Pour calculer Fact(3) on passe par les appèls suivants :

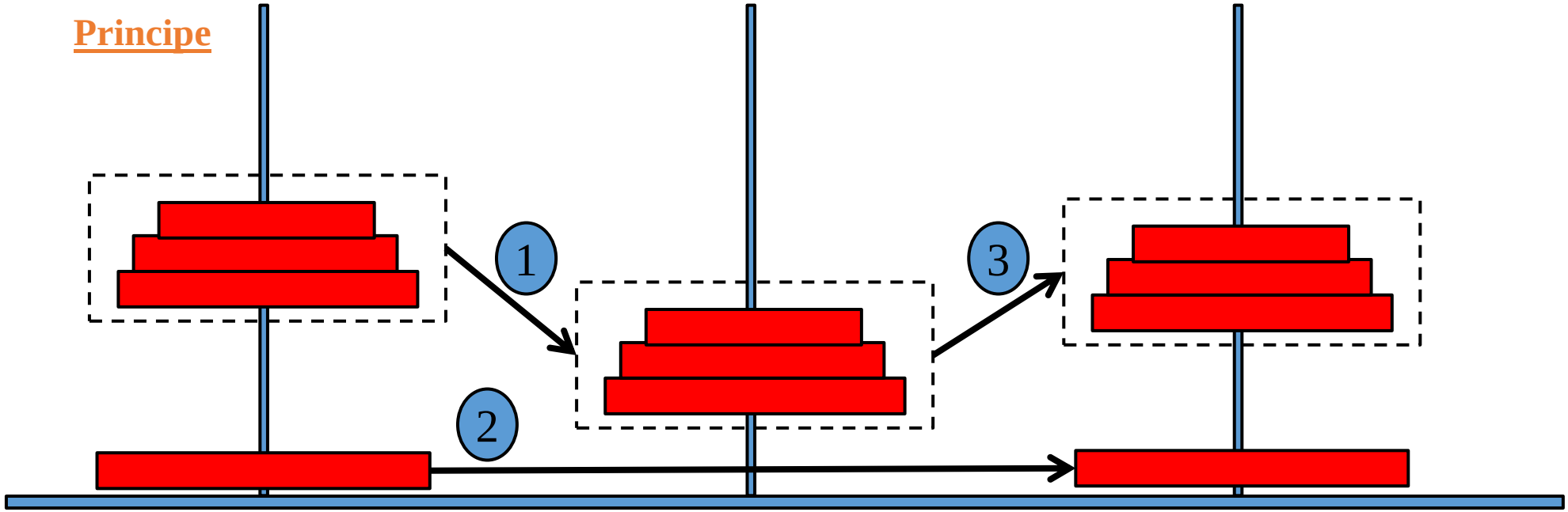
- **Fact(3) = 3 * fact(2)**
- **Fact(2) = 2* Fact(1)**
- **Fact(1)=1* Fact(0)**
- **Fact(0)=1**

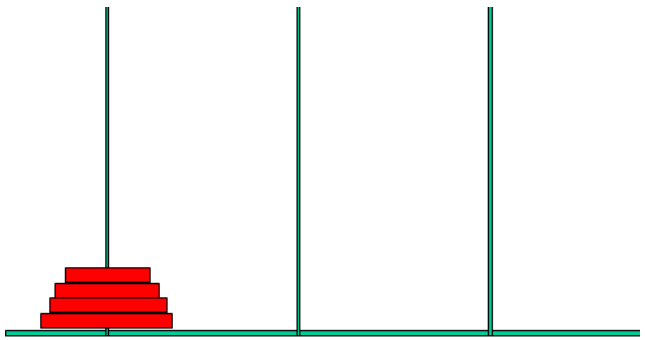
```
long Fact (int  n ){
   if( n= = 0 )
        return 1 ;
   else
        return n * Fact(n-1)
}
```

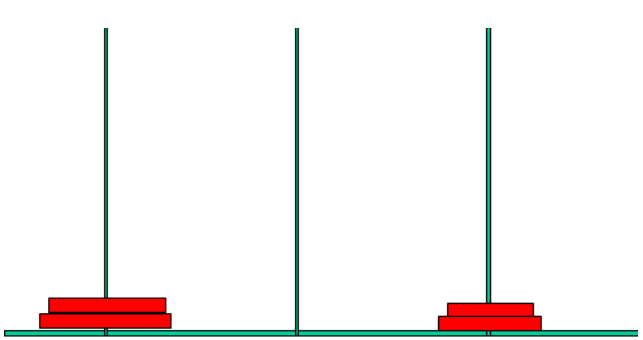
Exemple 2 Jeu de Hanoi qui consiste à déplacer un ensemble de disques rangés en ordre croissant des rayons vu du bas vers le haut dans une tige 1 vers une tige 2 en respectant les règles suivantes :

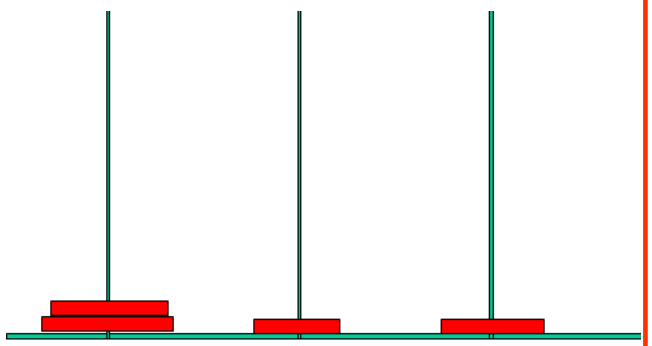
- On ne peut déplacer qu'un seul disque à la fois,
- On doit respecter l'ordre des rayons c'est-à-dire qu'on n'a pas le droit de poser un disque donné sur un disque de rayon plus petit
- On dispose d'une seul tige intermédiaire qui peut être utilisée pour atteindre le but recherché.
- Le score du jeu est inversement proportionnel au nombre de déplacements

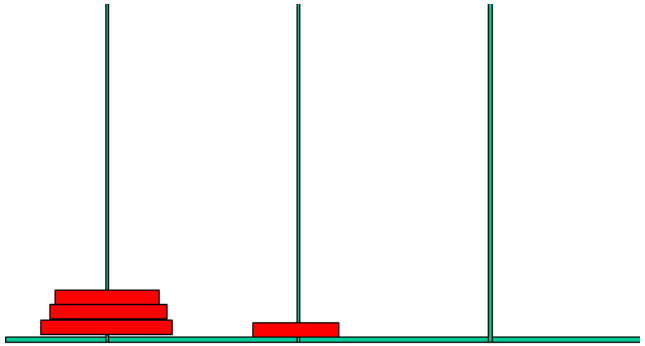
Principe

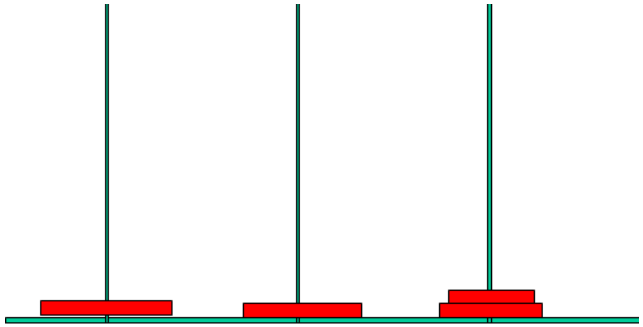


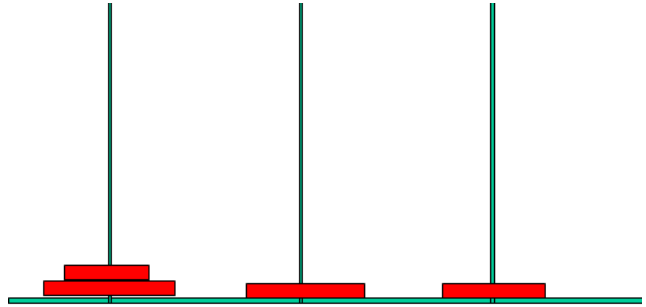


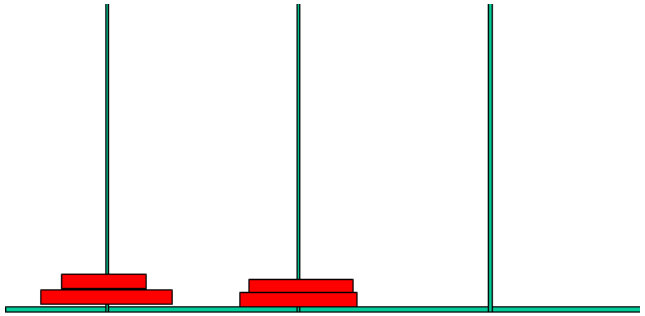


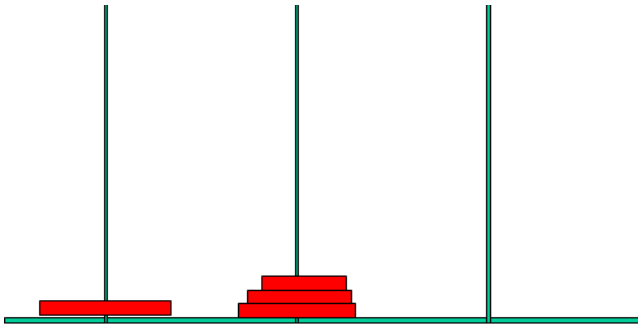


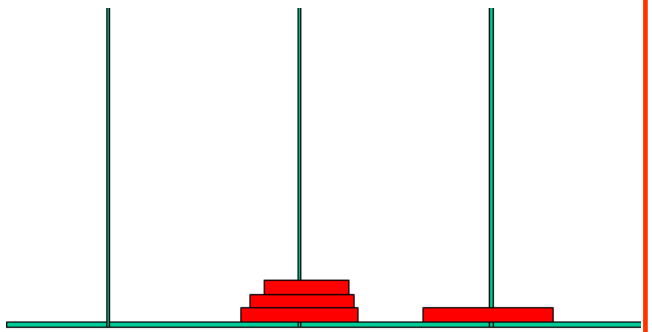


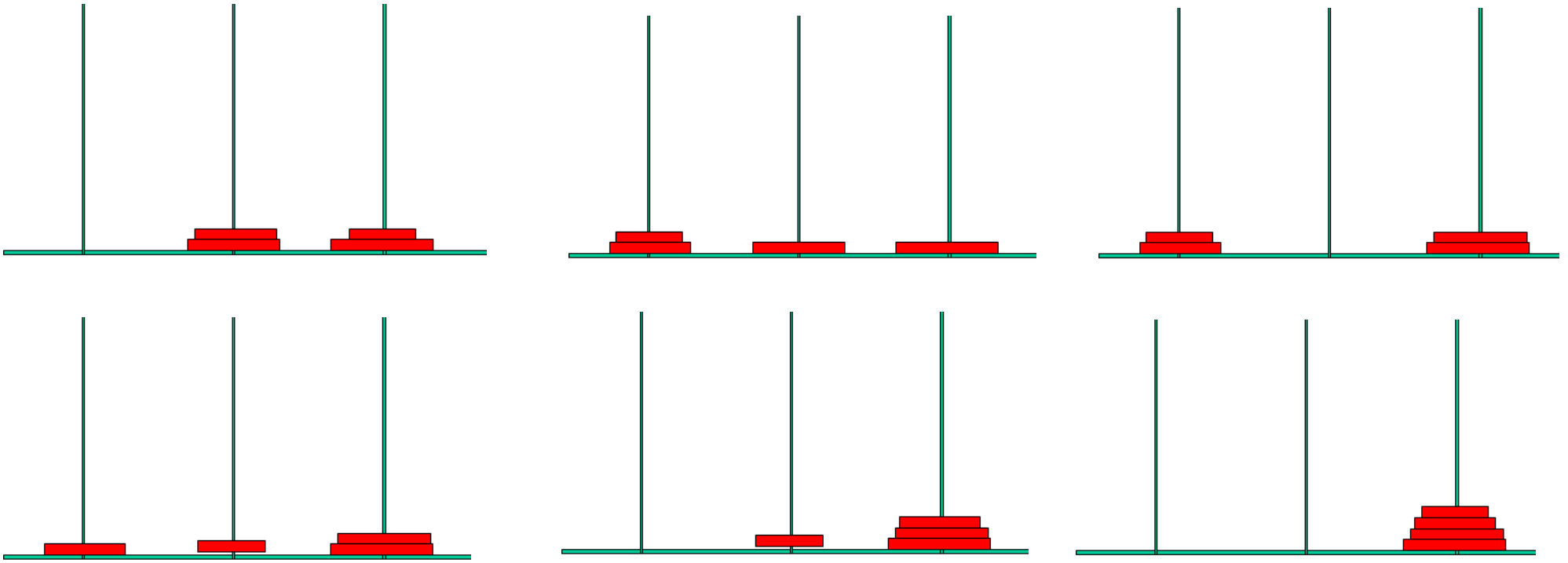












Solution

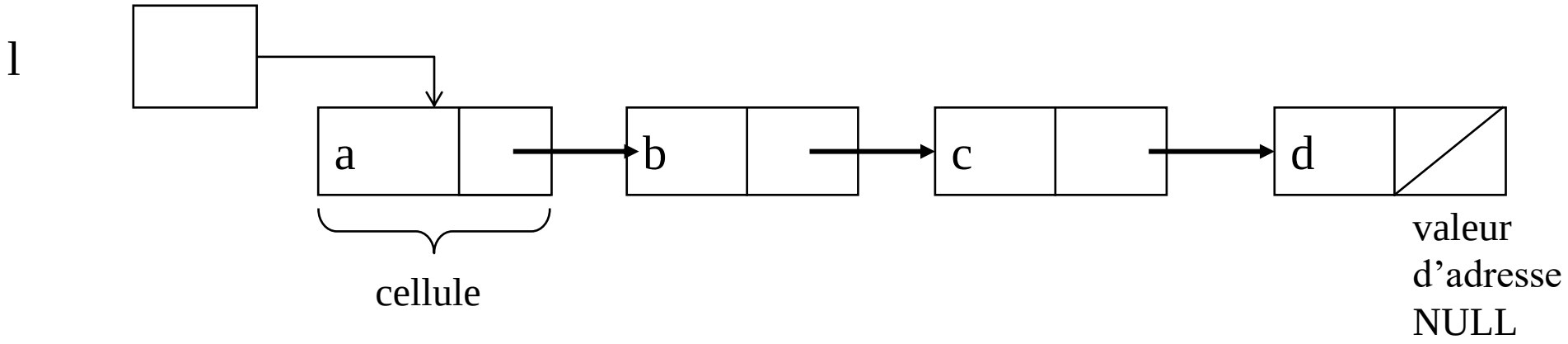
```
void Deplacer(int NbreTig, int NumTigSrc, int NumTigDest){
        if(NbreTig==1) DeplacerDisque(NumTigSrc,NumTigDesti);
        else{
                Deplacer(NbreTig-1,NumTigSrc,6-NumTigSrc-NumTigDest);
                DeplacerDisque(NumTigSrc,NumTigDest);
                Deplacer(NbreTig-1,6-NumTigSrc-NumTigDest,NumTigDesti);}
}
```

8. Applications :

8.1. Liste linéaire chaînée

Ecrire une application C qui gère une liste chaînée d'entiers par les opérations suivantes :

- Insertion en tête
- Suppression de l'élément en tête
- Affichage de la liste



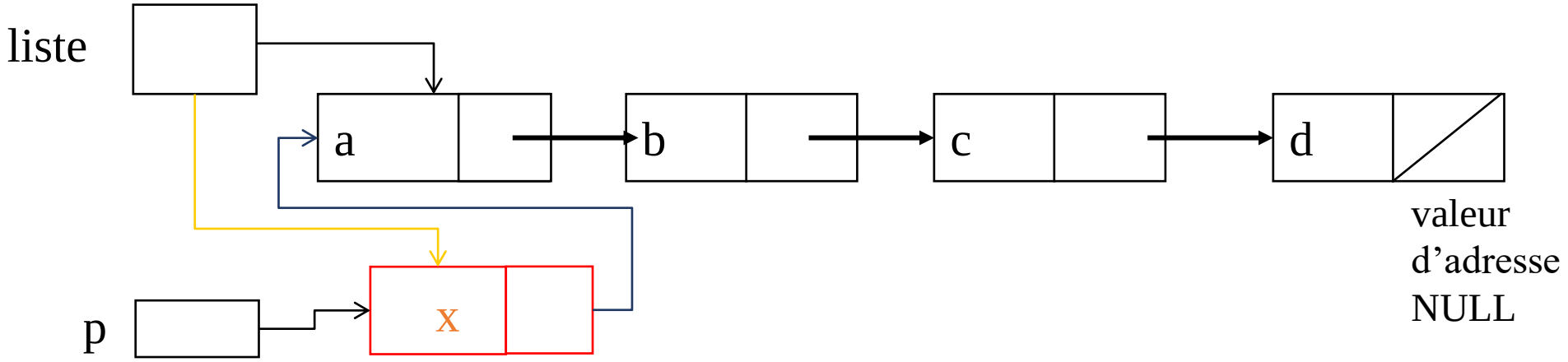
La liste est définie par l'adresse de sa première cellule

```
typedef struct tcellule{
        int  x ;
        struct tcellule *suivant;
}Tcellule;
typedef  Tcellule  *TListe;
```

opération insertion

Pour insérer une cellule en tête il faut :

- **Créer la cellule qu'il faut pointée par un pointeur p**
- **Définir la valeur x de la cellule p**
- **Faire pointer le suivant de p sur la tête de la liste**
- **Modifier la liste pour qu'elle pointe sur la cellule p**




```
void insertiontete(tliste *ptrList  , int e){
      Tliste p=(tliste)malloc(sizeof(tcellule));
      p→x=e;
      p→suivant=*ptrlist;
      *ptrList=p;
}
```

8.2. Polynomes

Type tpolynome

```
typedef struct {
    int degre ;
    double *coefficients ;
}tpolynome ;
```

Fonction sommePoly

```
tpolynome *sommePoly(tpolynome *p, tpolynome *q){
    Tpolynome *s=(tpolynome *)malloc(sizeof(tpolynome));
    int i, min=p→degre<q→degre?p→degre:q→degre;
    S→degre=p→degre>q→degre?p→degre:q→degre;
    S→coefficients=(double *)malloc((s→degre+1)*sizeof(double));
    for(i=0;i<min;i++)s→coefficients[i]=p→coefficients[i]+q→coefficients[i];
    for(;i<=p→degre;i++)s→coefficients[i]=p→coefficients[i];
    for(;i<=q→degre;i++)s→coefficients[i]=q→coefficients[i];
    return s;
}
```

Fonction produitPoly

```
tpolynome *produitPoly(tpolynome *p, tpolynome *q){
    Tpolynome *s=(tpolynome *)malloc(sizeof(tpolynome));
    int i, j,k,somDegre;
    somDegre=S→degre=p→degre+q→degre;
    s→coefficients=(double *)malloc((s→degre+1)*sizeof(double));
    for(k=0;k<=somDegre;k++){
        s→coefficients[k]=0;
        for(i=0;i<=k && i<=p→degre;i++)
            s→coefficients[k]+=p→coefficients[i]*q→coefficients[k-i];
    }
    return s;
}
```