# Langage de programmation C :

# les tableaux et les fonctions

# I- Les Tableaux

# Les tableaux

- Les tableaux à une dimensions
  - Un tableau à une dimension ne possède qu'un index. Un index est le nombre entre crochets qui suit le nom du tableau. Il indique le nombre d'éléments du tableau.
  - Déclarations

    **<type simple> Nom_du_tableau[nombre_elements];**
    - Type simple : définit le type d'élément que contient le tableau
    - Nom du tableau : est le nom qu'on donne au tableau, ce nom suit les mêmes règles qu'un nom de variable.
    - Nombre_elements : est une expression constante entière positive, qui exprime la taille du tableau.
  - **Exemple:**
    - char caractere[12];
    - int entier[10];
    - float reel[8];

# Les tableaux

- Les tableaux à une dimensions
  - Initialisation à la déclaration

    Il est possible d'initialiser le tableau à la définition :

**<type > Nom_du_tableau[nombre_elements]={c1,c2,…cn};**

Ou c1, c2,…cn sont des constantes dont le nombre ne doit pas dépasser le nombre _elements.

  - Exemple

```
#define  N_mois 12
   int tableau[N_mois];
   int  tab[4] ={100,200,300,400;}\\ initialisation
   char voyelles[6]={'a','e,'i','o','u','y'};
```

# Les tableaux

- Les tableaux à une dimensions
  - Accès aux composantes d'un tableau
    - Pour accéder à un élément du tableau, il suffit de donner le nom du tableau, suivi de l'indice de l'élément entre crochets :

      Nom_du_Tableau [indice]
    - Où indice est une expression entière positive ou nulle.
    - Un indice est toujours positif ou nul ;
    - L'indice du premier élément du tableau est 0 ;
    - L'indice du dernier élément du tableau est égal au nombre d'éléments – 1.
  - Exemples :
    - short A[5] = {12 , 23 , 34 , 45 , 56}; // A[0] = 12 ; A[1] = 23 ; A[2] = 34 ;

      // A[3] = 45 ;

      A[4] = 56 ;
    - for( i = 0 ; i< 5 ; i++ ) //Affichage des éléments

      printf(''A[%d] = %d \t'',i,A[i]); // du tableau A

# Les tableaux

- Les tableaux à une dimensions
  - Remarque
    - Chaque élément ( TAB[i] ) d'un tableau ( int TAB[20] ) est manipulé comme une simple variable, on peut :
      - la lire : scanf("%d", &TAB[i] ); *// TAB[i] sera initialisé par un*
        // entier saisi depuis le clavier
      - l'écrire : printf("TAB[%d] = %d", i , TAB[i] ); *//Le contenu de TAB[i]*
        //sera affiché sur écran
      - la passer en argument à une fonction : TAB[i] = pow(TAB[i],2);
        // = TAB[i] $^2$
    - Pour initialiser un tableau (TAB1) par les éléments d'un autre tableau (TAB2) : évitez d'écrire TAB1 = TAB2 (incorrect)
    - On peut par exemple écrire :

```
for( i = 0 ; i < taille_tableau ; i++ )
        TAB1[i] = TAB2[i];
```

    -

# Les tableaux 2D

- Les tableaux à deux dimensions

  - En C, un tableau multidimensionnel est considéré comme un tableau dont les éléments sont eux même des tableaux.

  - Un tableau à deux dimensions se déclare donc de la manière suivante :

    **int mat[10][20];**

  - En faisant le rapprochement avec les mathématiques, on peut dire que mat est une matrice de 10 lignes et de 20 colonnes.

# Les tableaux 2D

- Les tableaux à deux dimensions
  - On accède à un élément du tableau par l'expression mat[i][j].
  - Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

```
#include <stdio.h>
#define L 3 //nombre de lignes
#define C 2 //nombre de colonnes
int tab[L][C] = {{1, 2}, {14, 15}, {100, 200}};
main() {
int i, j;
for (i = 0 ; i < L; i++) {
      for (j = 0; j < C; j++)
          printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);}}
```

$$\texttt{tab} = \begin{pmatrix} 1 & 2 \\ 14 & 15 \\ 100 & 200 \end{pmatrix}$$

# Les tableaux

- **Modification des éléments d'un tableau passé en paramètre**

```
#include <stdio.h>
void print_tab(int tab[], int nb_elem) {...}
/* incrémente chaque composantes du tableau */
void incr_tab(int tab[], int nb_elem) {
int i;
for (i=0; i < nb_elem; i++) tab[i]++;}
#define TAILLE 4
 main() {
int t[TAILLE] = {1, 2, 3, 4};
incr_tab(t, TAILLE);
print_tab(t, TAILLE);
}
```

# II- Les fonctions

# Définition : Syntaxe

type identificateur ( liste de-déclarations-de-paramètres )
{
    liste-de-déclarations$_{option}$(optionnel)
    liste-d'instructions
}

- type identificateur ( liste-de-déclarations-de-paramètres );
  - ☛porte le nom de **prototype de fonction**.

# Définition : Sémantique

- type :est le type de la valeur rendue par la fonction ;
- identificateur :est le nom de la fonction ;
- Liste de déclarations-de-paramètres : est la liste (séparés par des virgules) des déclarations des paramètres formels.
- La liste-de-déclarations$_{option}$ : permet si besoin, de déclarer des variables qui seront locales à la fonction, elles seront donc inaccessibles de l'extérieur.
- La liste-d'instructions: est l'ensemble des instructions qui seront exécutées sur appel de la fonction. Parmi ces instructions, il doit y avoir au moins une instruction du type : **return expression ;**

# Définition: Appel d'une fonction

- Syntaxe :
  - Expression :

    ➔ identificateur ( liste-d'expressions );
- Sémantique :
  - Les expressions de liste-d'expressions sont évaluées, puis passées en tant que paramètres effectifs à la fonction de nom identificateur, qui est ensuite exécutée. La valeur rendue par la fonction est la valeur de l'expression appel de fonction.

# Exemple de définition et d'utilisation d'une fonction en C

```
/*************** la fonction fexple ****************/
float fexple (float x, int b, int c){
    float val ; /* déclaration d'une variable "locale" à fexple
   val = x * x + b * x + c ;
   return val ;
}
```

| float | fexple | (float x, | int b, | int c) |
|---|---|---|---|---|
| type de la "valeur de retour" | nom de la fonction | premier argument (type float) | deuxième argument (type int) | troisième argument (type int) |

# Exemple de définition et d'utilisation d'une fonction en C

```
#include <stdio.h>

        /***** le programme principal (fonction main) *****/
 float fexple (float, int, int) ; /* prototype de la fonction fexple */
 main(){
float x = 1.5 ;
   float y, z ;
   int n = 3, p = 5, q = 10 ;
        /* appel de fexple avec les arguments x, n et p */
   y = fexple (x, n, p) ;
   printf ("valeur de y : %e\n", y) ;
        /* appel de fexple avec les arguments x+0.5, q et n-1 */
   z = fexple (x+0.5, q, n-1) ;
  printf ("valeur de z : %e\n", z) ;
}
/* déclaration de fonction fexple */
```

# Exemple de définition et d'utilisation d'une fonction en C

```
/* définition de la fonction */

long cube(long x) {      /* en-tête de la fonction cube*/

   long  x_cube;  /* variable "locale" à la fonction cube */
    x_cube= x*x*x;
    return x_cube;
}
```

# Exemple de définition et d'utilisation d'une fonction en C

```
#include<stdio.h>

        /* déclaration de la fonction cube*/
long cube(long );    /* prototype de la fonction cube */
  /*****************************************************/
main(){
  long input, reponse;
  printf("entrez une valeur entière : ");
  scanf("%ld",&input);
     reponse=cube(input);
      printf("\n\n le cube de %ld est %ld\n", input,
 reponse);
}
```

# Exercice

***Quels seront les résultats fournis par ce programme :***

```
int arrondi (float) ; /* prototype de la fonction arrondi */
main(){
  float v1 = 1.6, v2 = 2.8 ;
   int p ;
   p = arrondi (v1) ; printf ("%d\n", p) ;
   p = arrondi (v2) ; printf ("%d\n", p) ;
   printf ("%d %d\n", arrondi(v1+v2), arrondi(v1) + arrondi(v2) ) ;
}
int arrondi (float r)
  { float y ;int n ;
     y = r + 0.5 ;
     n = y ;
     return n ;
}
```

# Autre exemple

*/*définition d'une fonction, nommée max, qui fournit en résultat la plus grande des trois valeurs entières reçues en paramètres*/*

```
int max (int a, int b, int c){
   int m ;
    m = a ;
    if ( b>m ) m = b ;
    if ( c>m ) m = c ;
    return m ;
}
```

# Autre exemple

```
            /*programme principal*/
int max (int, int, int) ; /* prototype de notre fonction max */
    main(){

     int n, p, q, m ;
     n = 3 ; p = 5 ; q = 2 ;
     m = max (n, p, q) ;
     printf ("max de %d %d %d : %d\n", n, p, q, m) ;
     m = max (5*n, n+p, 12) ;
     printf ("valeur : %d\n" , m) ;
}
```

# Exemple de fonction sans résultat

Soit la définition de fonction nommée optimist :

```
void optimist (int nfois){
  int i ;
    for (i=0 ; i<nfois ; i=i+1)
    printf ("il fait beau\n") ;
}
```

- Son en-tête montre qu'elle comporte un paramètre entier (nommé *nfois*) ; *cette fois, il est* précédé du mot ***void*** *qui indique que la fonction ne fournit pas de résultat*.
- Si l'on examine les instructions du corps de la fonction, on constate qu'elles affichent un certain nombre de fois le même texte : *il fait beau. Pour effectuer son travail, notre fonction a eu besoin d'une variable* locale (*i*).

# Exemple de fonction sans résultat

- **Remarques:**
  - Aucune instruction return ne figure dans la définition de notre fonction.
  - y = optimist (k) ;  /* incorrect */
  - l'appel de cette fonction ce fait à l'aide d'une instruction de la forme :

```
optimist (k) ;   /* instruction simple provoquant
                    l'appel de optimist à laquelle on
                    transmet en paramètre, la valeur de k */
```

# Exemple de fonction sans résultat

```
void optimist (int) ;   /* prototype de la fonction optimist */
main(){
int n = 2, p = 1 ;
   optimist (n) ;
   optimist (p) ;
   optimist (n+p) ;
}
void optimist (int nfois)
{ int i ;
  for (i=0 ; i<nfois ; i=i+1)
  printf ("il fait beau\n") ;
}
```

# Le cas des fonctions sans paramètres

- Si une fonction ne possède aucun paramètre, son en-tête et sa déclaration (prototype) doivent comporter le mot ***void***, *à la place de la liste des paramètres.*
- *Exemple :*
  - int fexple1 (void)
  - Sa déclaration (prototype) serait:

    int fexple1 (void) ;
- L'en-tête d'une fonction ne recevant aucun paramètre et ne fournissant aucun résultat est:
  - void fexple2 (void)
- Son prototype est:
  - void fexple2 (void) ;

# Le cas des fonctions sans paramètres

- L'appel d'une fonction sans paramètres doit comporter des parenthèses vides.
  - Exemple:
  - L'appel de *fexple1 s'écrira :* ***fexple1()***
  - *et non simplement :* ***fexple1***
  - Exemple :

```
#include<stdio.h>
void affiche(void);
main(){
  affiche();
  }
void affiche(void){
printf(''bonjour'');
  }
```

# Règles

- Arguments muets et arguments effectifs
  - Les noms des arguments figurant dans l'en-tête de la fonction se nomment des « arguments muets », ou encore « arguments formels » ou « paramètres formels » (de l'anglais : *formal parameter). Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle* doit faire.
  - Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des « arguments effectifs » (ou encore « paramètres effectifs »).

# Règles: L'instruction return

- L'instruction return peut mentionner n'importe quelle expression. Ainsi, nous pourrons définir la fonction fexple d'une manière plus simple :
  - ```
    float fexple (float x, int b, int c){
     return (x * x + b * x + c) ;}
    ```
- L'instruction return peut apparaître à plusieurs reprises dans une fonction, comme dans cet autre exemple :
  - ```
    double absom (double u, double v){
      double s ;
       s = a + b ;
       if (s>0) return (s) ;
      else        return (-s);
    }
    ```

# Règles : Initialisation des variables locales

- Les variables locales peuvent être initialisées lors de leur déclaration. Dans ce cas, il faut savoir que la valeur indiquée est placée dans la variable, non pas au moment de la compilation, mais à chaque appel de la fonction. Par exemple, avec cette définition :

```
void affiche (void){
    int n = 10 ;
    printf ("%d", n) ;
    n = n + 1 ;
}
```

- on obtiendra l'affichage de la valeur 10, à chaque appel de affiche.

# Une fonction peut en appeler une autre

- Rien n'empêche qu'une fonction appelle, à son tour, une autre fonction, comme dans ce canevas :

```
main(){
 int f1 (float) ; /* prototype de f1 */
     .....
    f1 (...) ; /* appel de f1 */
   .....
}
int f1 (float x){
  void f2 (int) ; /* prototype de f2 */
.....
Return(...);}
void f2 (int n) {
.....
}
```

# En langage C, les paramètres sont transmis par valeur

```
void echange (int a, int b) ;
main(){
int n=10, p=20 ;
printf ("avant appel : %d %d\n", n, p) ;
echange (n, p) ;
printf ("après appel : %d %d", n, p) ;
}
void echange (int a, int b){
   int c ;
   printf ("début echange : %d %d\n", a, b) ;
   c = a ; a = b ; b = c ;
  printf ("fin echange : %d %d\n", a, b) ;
}
```

```
avant appel :      10 20
début echange :  10 20
fin echange :       20 10
après appel :      10 20
```

# Passage de tableau en paramètre

- Tableau à une dimension de taille fixe

```
void affiche (int  [5]) ; /* prototype de la fonction  affiche */
main(){
int i ;
 int t[5] = { 1, 1, 1, 1, 1 } ; /* initialisation du tableau */
printf ("tableau t avant appel de affiche : ") ;
 for (i=0 ; i<5 ; i=i+1)   printf ("%d ", t[i]) ;
 printf ("\n") ;
  affiche (t) ; /* appel de affiche, à laquelle on transmet en paramètre le
  tableau t */
  printf ("tableau t après appel de affiche : ") ;
  for (i=0 ; i<5 ; i=i+1)    printf ("%d ", t[i]) ;
}
void affiche (int v[5]){
   int i ;
  for (i=0 ; i<5 ; i=i+1)
  v[i] = i+1 ;}
```

# Passage de tableau en paramètre

- Tableau à deux dimensions de taille fixe

```
void mat (int t[5][4]){
  int i, j ;
  for (i=0 ; i<5 ; i++)
    for (j=0 ; j<4 ; j++)
                t[i][j] = 1 ;
}
```

Voici quelques exemples d'utilisation de cette fonction :

```
main (){
  int tab [5] [4] ;
  mat (tab) ;
}
```

# Fonctions prédéfinies : math.h

- Pour utiliser les fonctions de cette librairie, il faut inclure la librairie par la directive #include <math.h>

## .1 Fonctions trigonométriques et hyperboliques

| Fonction | Sémantique | Fonction | Sémantique |
|---|---|---|---|
| acos | arc cosinus | cos | cosinus |
| asin | arc sinus | sin | sinus |
| atan | arc tangente | tan | tangente |
| cosh | cosinus hyperbolique | sinh | sinus hyperbolique |
| tanh | tangente hyperbolique | atan2 | arc tangente |

# Fonctions prédéfinies : math.h

## .2 Fonctions exponentielles et logarithmiques

| Fonction | Sémantique |
|---|---|
| exp | exponentielle |
| frexp | étant donné x, trouve n et p tels que x = n * 2p |
| ldexp | multiplie un nombre par une puissance entière de 2 |
| log | logarithme |
| log10 | logarithme décimal |
| modf | calcule partie entière et décimale d'un nombre |

## .3 Fonctions diverses

| Fonction | Sémantique |
|---|---|
| ceil | entier le plus proche par les valeurs supérieures |
| fabs | valeur absolue |
| floor | entier le plus proche par les valeurs inférieures |
| fmod | reste de division |
| pow | puissance |
| sqrt | racine carrée |