Les fonctions

Déclaration d'une fonction

<TypeRés> <NomFonct> (<TypePar1> <NomPar1>, <TypePar2> NomPar2>, ...) {

<déclarations locales> <instructions> }

Exemple

```
int MAX(int N1, int N2){
    if (N1>N2)
        return N1;
        else
        return N2;
    }
```

Remarques

- ✓ Une fonction peut fournir comme résultat:
  - ✓ un type arithmétique,
  - ✓ une structure (définie par **struct**)
  - ✓ un pointeur,
  - ✓ **void** (la fonction correspond alors à une 'procédure').
- ✓ Une fonction **ne peut pas** fournir comme résultat des tableaux, des chaînes de caractères ou des fonctions.
- ✓ Le type par défaut est **int**;
- ✓ Il est interdit de définir des fonctions à l'intérieur d'une autre fonction.
- ✓ Chaque fonction doit être déclarée ou définie avant d'être appelée.

Déclaration d'une fonction

- ❑ **Déclaration explicite par le prototype (entête)**

  <TypeRés> <NomFonct> (<TypePar1>, <TypePar2>, ...);

- ❑ **Déclaration implicite par définition (implémentation)**

Portée de variables et de fonctions

- Un bloc d'instruction est encadré d'accolade { } est composé de 2 parties :
  - Déclarations des variables locales
  - Instructions

➔ **Variables locales**
➔ **Variables globales**

Modes de passage de paramètres

➢ Passage par valeur (par copie)

➢ Passage par variable (par adresse)

Exemple:
La fonction Permuter

Passage de l'adresse d'un tableau

Exemple

La fonction LIRETAB lit N données pour un tableau du type **int** et les mémorise à partir de l'adresse indiquée par le pointeur PTAB.

PTAB et N sont fournis comme paramètres.

```
void LIRE_TAB(int N, int *PTAB)
{
printf("Entrez %d valeurs : \n", N);
while(N)  {
   scanf("%d", PTAB++);
   N--  }
}
```

Exercices

1. Ecrire deux fonctions qui calculent la valeur X^N pour une valeur réelle X (type **double**) et une valeur entière positive N :

a) EXP1 retourne la valeur X^N comme résultat.

b) EXP2 affecte la valeur X^N à X.

Ecrire un programme qui teste les deux fonctions à l'aide de valeurs lues au clavier.

2. Ecrire la fonction SOMME_TAB qui calcule la somme des N éléments d'un tableau TAB du type **int**. N et TAB sont fournis comme paramètres; la somme est retournée comme résultat du type **long**.

Correction

1. a.

```
int  EXP1(int x, int N){
        int s = 1;
        while(N){
              s*=x;
              N--;
        }
        return s;
}
```

Rm la valeur de N ne changera pas à la sortie de la fonction car passage par valeur

➔ Appel

```
void main() {
        int a = 3, b;
        b=EXP1(a,2);// b prend le carré de 3
}
```

Correction

1. b.

```
void EXP2(int *x, int N){
        int s = 1;
        while(N){
              s*=*x;
              N--;
        }
        *x=s;
}
```

Rm on choisit le passage par adresse pour x pour pouvoir changer sa valeur à la sortie de la fonction

➔ Appel

```
void main() {
        int a = 3;
        EXP2(&a,2);// a prend son carré
}
```

Correction

2. On choisira de représenter le tableau selon le formalisme pointeur

```
long  Somme_Tab(int * t, int N) { //Calculer la somme
        int s=0, *p;
        for (p=t; p<t+N; p++)
                s+=*p;
        return s;
}
void Lire_Tab(int * t, int N) { //Lire le tableau
         int *p;
        for (p=t; p<t+N; p++)
                scanf(« %d », p);
}
```

➔ *Appel*

```
void main()
{
        int *tab, x;
        scanf(« %d », &x); //Lire le nombre d'éléments
        tab=(int*)malloc(x*sizeof(int)); //Alouer x nombres entiers
        Lire_Tab(tab, x);
        printf(« %d », Somme_Tab(tab,x)); //Afficher la somme du tableau
}
```

Les structures

12

Définition d'un nouveau type structure

```
struct complexe{
      float reel;
      float imag;
};
```

```
typedef struct {
      float reel;
      float imag;
} complexe;
```

```
Typedef struct _complexe{
      float reel;
      float imag; }
complexe;
```

Déclaration de variables structure

```
struct complexe
{
...
} ;
void main() {
int n;
struct complexe C1,*C2;
...
}
```

Ou bien

```
typedef struct
{
...
}complexe ;
void main() {
int n;
complexe C1,*C2;
...
}
```

Accès aux champs d'une structure

- Initialisation : Complexe Z={0.5, 2.0};
- Déclaration statique:

```
Complexe Z;
Z.Im = 0.5;
Z.Re = 2.0;
```

- Déclaration dynamique:

```
Complexe *Z;
Z=(Complexe*)malloc(sizeof(Complexe));
Z-> Im = 0.5;
Z-> Re = 2.0;
```

Exercices

1. Ecrire une fonction qui affiche un nombre complexe passé en argument.
2. Ecrire une fonction qui retourne le module d'un nombre complexe passé en argument.
3. Ecrire une fonction qui affiche N nombres complexes stockés dans un tableau.
4. Ecrire une fonction qui retourne la somme des modules de N complexes stockés dans un tableau.

Correction

```
1. void  Affiche_Complexe (Complexe C){
    printf(« %f %f », C.Im , C.Re);
}
2. double Module(Complexe C){
    return (sqrt(C.Im*C.Im+C.Re*C.Re)); //
}
3.  //Version statique
void Affiche(Complexe t[], int N) {
    int i;
    for (i=0; i<N; i++)
         Affiche_Complexe(t[i]);
}
```

$$Module = \sqrt{C.Im^2 + C.Re^2}$$

Correction

```
//Version dynamique
void Affiche(Complexe* t, int N) {
    Complexe* p;
    for (p=t; p<t+N; p++)
         Affiche_Complexe(*p);
}
```

➔ Appel

```
void main(){
        Complexe *tab,*p;
        int x;
        scanf(« %d », x);
tab= (Complexe*)malloc(x*sizeof(Complexe));
//Lire les éléments du tableau
//Evidemment on peut écrire une fonction pour ça
for (p=tab; p<tab+x; p++)
    scanf(« %f %f », &(p->Re), &(p->Im));
//Afficher le tableau
Affiche(tab, x);
//Afficher les modules
for (p=tab; p<tab+x; p++)
    printf(« %f», Module(*p));
}
```

4.

```
double Somme_Modules(Complexe* t, int N) {
    double S=0;
    Complexe* p;
    for (p=t; p<t+N; p++)
        S + = Module (*p);

    return (S);
}
```