CHAPITRE 4 : LES FONCTIONS

I. Introduction :

Lorsqu'on développe un programme et que le problème à résoudre est complexe, le nombre d'instruction devient vite important : une seule application peut facilement contenir plusieurs dizaines de milliers de lignes de code.

Pour pouvoir travailler dans un tel labyrinthe d'instructions, il est nécessaire de l'organiser : c'est le premier rôle des ***fonctions*** qui permettent de regrouper sous un même nom des instructions agissant dans un même but. Il suffit alors d'appeler la fonction ou la procédure par son nom pour exécuter les instructions correspondantes, sans avoir à les réécrire.

Les fonctions permettent donc de diviser un problème en plusieurs problèmes de complexité moindre : le programme est structuré de manière descendante, du problème le plus général aux problèmes les plus spécifiques, ce qui facilite sa conception et sa lisibilité. *Cette démarche est appelée "**Programmation structurée**".*

Lorsqu'on développe un programme, on a souvent besoin de répéter les mêmes suites d'instructions pour réaliser la même opération à différents moments de l'exécution du programme : Calcul d'un maximum, résolution d'une équation, affichage d'une page à l'écran... C'est le deuxième rôle des fonctions qui sont des regroupements d'instructions pouvant être appelés plusieurs fois à partir du programme principal ou à partir d'autres fonctions en recevant à chaque fois des paramètres ayant une valeur différente. Les fonctions sont donc des moyens de réutilisation de portions de code.

II. Définition des fonctions :

II-1. Déclaration:

Une fonction doit être déclarée (définie) avant toute utilisation. La déclaration, qui consiste à expliciter la fonction, fonctionne de la même manière que dans le cas des variables, et permet au compilateur de savoir quels sont les arguments que prend la fonction, leur type, et le type de valeur qu'elle retourne.

Syntaxe :

```
type   nom-fonction ( type-1 arg-1,..., type-n arg-n)
{
        déclarations de variables locales
        liste d'instructions
}
```

La première ligne de cette déclaration est l'en-tête de la fonction. Dans cet en-tête, **type** désigne le type de la fonction, c'est-à-dire le type de la valeur qu'elle retourne. Contrairement à d'autres langages, il n'y a pas en C de notion de procédure ou de sous-programme. Une fonction qui ne renvoie pas de valeur est une fonction dont le type est spécifié par le mot-clef ***void***. Les arguments de la fonction sont appelés paramètres formels, par opposition aux paramètres effectifs qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction.

Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'instruction de retour à la fonction appelante, **return**, dont la syntaxe est : **return(expression);**

La valeur de ***expression*** est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type **void**), sa définition s'achève par **return;**

Plusieurs instructions **return** peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier **return** rencontré lors de l'exécution.

Exemple :

- Une fonction qui calcule la factorielle de son argument :

```
int factorielle (int n)
{
        int resultat=1, i;
        for (i=1;i<=n;i++)
                resultat *= i;
        return (resultat);

}
```

- Une fonction récursive qui calcule la puissance a^n :

```
int puissance (int a, int n)
{
        if (n == 0)
                return(1);
        else
        return(a * puissance(a, n-1));
}
```

- La fonction ***main()*** que doit comporter tout programme en C :

```
int main( ) {...}
```

- La fonction ***main*** ne prend (ici) aucun argument, et retourne un **entier**.
- **{...}** représente les instructions définissant la fonction.

II-2. Prototype :

Il arrive souvent que l'on a besoin d'une fonction dont la déclaration (définition) se trouve soit dans un fichier différent, soit dans plus loin dans le programme (après la fonction **main**). Il est alors nécessaire d'utiliser un prototype : il s'agit d'une instruction correspondant à une déclaration restreinte de la fonction, et qui indique :

– le nombre d'arguments qu'elle prend,
– le type de ses arguments,
– sa valeur de retour.

Exemple :

- le prototype de notre fonction factorielle est : **int factorielle (int);**
- le prototype de notre fonction puissance est : **int puissance (int , int) ;**

Tout comme pour les déclarations de variables, les prototypes de fonction prennent un point-virgule à la fin.

Remarque : la directive #include <stdio.h> permet d'inclure les prototypes des fonctions de la bibliothèque stdio dans l'en-tête d'un fichier source.

II-3. Appel d'une fonction :

Pour exécuter une fonction, il suffit de faire appel à elle en écrivant son nom (une fois de plus en respectant la casse) suivi d'une parenthèse ouverte (éventuellement des arguments) puis d'une parenthèse fermée :

nom-fonction(para-1,para-2,...,para-n) ;

L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions. La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation.

II-4. Un exemple complet :

```
#include <stdio.h>
int Max( int i, int j )
   {
        if ( i > j )        return i ;
        else     return j ;
   }
main()
   {
        int i, j, k, l, m , n ;
        i = 1 ;
        j = 2 ;
        k = 3 ;
        l = Max( i, j );
        m = Max( i, Max(j, k) ) ;
        n = j * Max( j,  k) ;
        printf( "L = %d, M = %d, N = %d\n", L, M, N ) ;
    }
```

III. Modes de transmission des paramètres :

En informatique, il existe deux modes de transmission des paramètres à une fonction :

III-1. Par Valeur :

Lors de l'appel, la valeur du paramètre effectif est recopiée dans le paramètre formel correspondant. Le paramètre effectif n'est donc jamais affecté par les modifications du paramètre formel, puisqu'il n'existe alors aucun lien entre eux.

```
#include <stdio.h>
void inverse(int a, int b)
   {
        int aux;
        aux = a ;
        a = b ;
        b = aux ;
   }
main ()
   {
        int un  = 1 ;
        int deux  = 2 ;
        inverse( un, deux ) ;   /*  La valeur contenu dans un et deux est  recopiée dans
                                                les paramètres formels a et b */
        printf( " un = %d deux = %d\n",  un, deux )
   }
/* => affiche : un = 1 et deux = 2 */
```

III-2. Par Adresse :

Lors de l'appel, l'adresse du paramètre effectif est recopiée dans l'adresse du paramètre formel correspondant. Le paramètre formel représente alors le même objet en mémoire que le paramètre effectif : toute modification de l'un est une modification de l'autre.

```
#include <stdio.h>

void inverse(int * a, int *b)   /* Cette fonction  reçoit deux pointeurs sur des entiers */
   {
        int aux ;
        aux = *a                /* la valeur de la variable pointée par a est mise dans aux */
        *a = *b ;               /* la valeur de la variable pointée par b est mise dans la
        *b = aux ;                                          variable pointée par a */
                              /* la valeur de aux est mise dans la variable pointée par b */
   }

main ()
   {
        int un  = 1 ;
        int deux  = 2 ;
        inverse( &un, &deux ) ; /*  Les adresses des variables un et deux sont
                                        recopiées dans les paramètres formels a et b */
        printf( " un = %d deux = %d\n",  un, deux )
   }

/* => affiche : un = 2 et deux = 1 */
```

III-3. Passage de tableaux en paramètres :

Il est possible de passer des tableaux en paramètres d'une fonction. Le passage d'un tableau en paramètre ne se fait jamais par valeur (donc toujours par adresse) : les valeurs contenues dans le tableau ne sont jamais recopiées, seul l'adresse du début de ce tableau est recopiée.

Pour indiquer qu'une fonction reçoit un tableau en argument, il suffit donc de donner dans la liste des arguments une variable de type tableau (inutile de préciser la taille dans ce cas : peut importe) ou de type pointeur sur un élément de ce tableau (Donner l'adresse d'un tableau, ou l'adresse de son premier élément est équivalent).

```
#include <stdio.h>

void affiche_tab( int t[], int taille)     /*  ou : void affiche_tab( int * t, int taille) */
   {
        int i ;
        for ( i=0 ; i<taille ; i++ )  printf ( "%d ", t[ i ] ) ;
   }

main()
   {
        int tableau[ 10 ] ;
        ...
        /* Appel de affiche_tab : */
        affiche_tab(tableau,10) ;        /* ou affiche_tab( &tableau[0] */
   }
```

Remarque : Un tableau passé en argument étant à priori de taille inconnue, il est généralement nécessaire de fournir aussi la taille de ce tableau en argument (ce qui permet de réaliser des fonctions sachant traiter toute taille de tableau).

IV. Durée de vie des variables :

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même durée de vie. On distingue deux catégories de variables.

- *Les variables permanentes (ou statiques)* : Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation.
- *Les variables temporaires :* Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction.

La durée de vie des variables est liée à leur portée, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

IV-1. Variables globales :

On appelle *variable globale* une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, **n** est une variable globale :

```
#include <stdio.h>
int n;       /* variable globale*/
void fonction(){
        n++;
        printf("appel numero %d\n",n);
 }
main()
 {
        int i;
        for (i = 0; i < 5; i++)
                fonction();
 }
```

La variable n est initialisée à zéro par le compilateur et il s'agit d'une variable permanente. En effet, le programme affiche :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

IV-2. Variables locales :

On appelle *variable locale* une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.

Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant :

```
#include <stdio.h>
int n = 10;   /* variable globale */
void fonction()
 {
        int n = 0;   /* variable locale  */
        n++;         /* incrémente la variable  locale  */
        printf("appel numero %d\n",n);  /* Affiche la variable  locale  */
 }
main()
 {
        int i;
        for (i = 0; i < 5; i++)
                fonction();
 }
```

Affiche :

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

V. Les fonctions récursives :

En C, toute fonction peut appeler toute fonction dont elle connaît le nom. En particulier, elle peut s'appeler elle-même. Il est donc possible d'écrire des fonctions récursives.
Prenons l'exemple le plus connu en matière de récursivité : la factorielle. Cette fonction factorielle peut s'écrire de la manière suivante :

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

Le résultat de chaque appel est stocké dans la pile. (PILE = zone mémoire accessible seulement en entrée (fonctionnement de type LIFO). LIFO = Last In First Out (dernier entré, 1er sorti)). Les appels s'arrêtent avec une « condition d'arrêt ». Les résultats empilés sont dépilés en ordre inverse.