CHAPITRE 8 : LES FONCTIONS

La structuration de programmes en sous-programmes se fait en C à l'aide de ***fonctions***. Nous avons déjà utilisé des fonctions prédéfinies dans des bibliothèques standard (**printf** de <*stdio*>, **strlen** de <*string*>, **pow** de <*math*>, etc.). Dans ce chapitre, nous allons découvrir comment nous pouvons définir et utiliser nos propres fonctions.

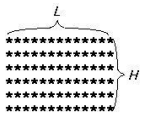
I) Modularisation de programmes :

1) Exemples de modularisation en C :

Le programme présenté ci-dessous donne un petit aperçu sur les propriétés principales des fonctions en C. Les détails seront discutés plus loin dans ce chapitre.

a) Exemple 1 : Afficher un rectangle d'étoiles

Le programme suivant permet d'afficher à l'écran un rectangle de longueur L et de hauteur H, formé d'astérisques '*' :



Implémentation en C

```
#include <stdio.h>

main()
    {
/* Déclaration des variables locales de main */
int L, H;

/* Prototypes des fonctions appelées par main */
void RECTANGLE(int L, int H);

/* Traitements */
printf("Entrer la longueur (>= 1): "); scanf("%d", &L);
printf("Entrer la hauteur (>= 1): "); scanf("%d", &H);

/* Afficher un rectangle d'étoiles */
RECTANGLE(L,H);
return (0);
    }
```

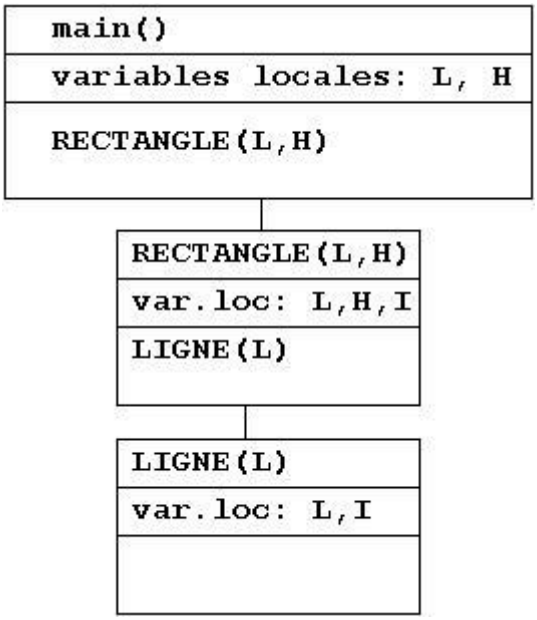
Pour que la fonction soit exécutable par la machine, il faut encore spécifier la fonction RECTANGLE :

```
void RECTANGLE(int L, int H)
{
/* Déclaration des variables locales */
int I;
/* Prototypes des fonctions appelées */
void LIGNE(int L);
/* Traitements */
/* Afficher H lignes avec L étoiles */
for (I=0; I<H; I++)
LIGNE(L);
}
```

Pour que la fonction RECTANGLE soit exécutable par la machine, il faut spécifier la fonction LIGNE :

```
void LIGNE(int L)
{
/* Affiche à l'écran une ligne avec L étoiles */
/* Déclaration des variables locales */
int I;
/* Traitements */
for (I=0; I<L; I++)
   printf("*");
printf("\n");
}
```

Schématiquement, nous pouvons représenter la hiérarchie des fonctions du programme comme suit :



II) La notion de blocs et la portée des identificateurs :

Les fonctions en C sont définies à l'aide de blocs d'instructions. Un bloc d'instructions est encadré d'accolades et composé de deux parties :

Exemple :

La variable d'aide I est déclarée à l'intérieur d'un bloc conditionnel. Si la condition (N>0) n'est pas remplie, I n'est pas défini. A la fin du bloc conditionnel, I disparaît.

```
if (N>0)
{
int I;
for (I=0; I<N; I++)
   ...
}
```

1) Variables locales :

Les variables déclarées dans un bloc d'instructions sont *uniquement visibles à l'intérieur de ce bloc*. On dit que ce sont des ***variables locales*** à ce bloc.

Exemple :

La variable NOM est définie localement dans le bloc extérieur de la fonction HELLO. Ainsi, aucune autre fonction n'a accès à la variable NOM :

```
void HELLO(void);
{
  char NOM[20];
  printf("Introduisez votre nom : ");
  gets(NOM);
  printf("Bonjour %s !\n", NOM);
}
```

Exemple :

La déclaration de la variable I se trouve à l'intérieur d'un bloc d'instructions conditionnel. **Elle n'est pas visible à l'extérieur** de ce bloc, ni même dans la fonction qui l'entoure.

```
if (N>0)
{
  int I;
  for (I=0; I<N; I++)
     ...
}
```

Attention !

Une variable déclarée à l'intérieur d'un bloc ***cache*** toutes les variables du même nom des blocs qui l'entourent.

Exemple :

Dans la fonction suivante,

```
int FONCTION(int A);
{
  int X;
  ...
  X = 100;
  ...
  while (A>10)
{
        double X;
        ...
        X*=A;
        ...
     }
}
```

la première instruction **X=100** se rapporte à la variable du type **int** déclarée dans le bloc extérieur de la fonction; l'instruction **X*=A** agit sur la variable du type **double** déclarée dans la boucle **while**. A l'intérieur de la boucle, il est impossible d'accéder à la variable X du bloc extérieur.

2) Variables globales :

Les variables déclarées au début du fichier, à l'extérieur de toutes les fonctions *sont disponibles à toutes les fonctions du programme*. Ce sont alors des ***variables globales***. En général, les variables globales sont déclarées immédiatement derrière les instructions **#include** au début du programme.

Attention !

Les variables déclarées au début de la fonction principale **main**
ne sont pas *des variables globales, mais elles sont locales à* **main *!***

Exemple :

La variable STATUS est déclarée globalement pour pouvoir être utilisée dans les procédures A et B.

```
#include <stdio.h>
int STATUS;

void A(...)
{
...
if (STATUS>0)
STATUS--;
else
...
...
}
```

```
void B(...)
{
...
STATUS++;
...
}
```

III) Déclaration et définition de fonctions :

En général, le nom d'une fonction apparaît à trois endroits dans un programme :

- lors de la **déclaration**
- lors de la **définition**
- lors de l'**appel**

1) Définition d'une fonction :

Dans la définition d'une fonction, nous indiquons :

- le nom de la fonction
- le type, le nombre et les noms des paramètres de la fonction
- le type du résultat fourni par la fonction
- les données locales à la fonction
- les instructions à exécuter

Définition d'une fonction en C :

```
<Type> <NomFonct> (<TypePar1> <NomPar1>, <TypePar2> <NomPar2>, ... )
   {
      <déclarations locales>
      <instructions>
   }
```

Remarquez qu'il n'y a pas de point virgule derrière la définition des paramètres de la fonction.

Attention !

*Si nous choisissons un nom de fonction qui existe déjà dans une bibliothèque, notre fonction **cache** la fonction prédéfinie.*

Type d'une fonction :

Si une fonction F fournit un résultat du type T, on dit que *'la fonction F est du type T'* ou que *'la fonction F a le type T'*.

Exemple :

La fonction MAX est du type **int** et elle a besoin de deux paramètres du type **int**. Le résultat de la fonction MAX peut être intégré dans d'autres expressions.

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

Exemple :

La fonction PI fournit un résultat rationnel du type **float**. La liste des paramètres de PI est déclarée comme **void** (vide), c'est-à-dire PI n'a pas besoin de paramètres et il faut l'appeler par : **PI()**

```
 float PI(void)
 {
   return 3.1415927;
 }
```

Rappel : main

La fonction principale **main** est du type **int**. Elle est exécutée automatiquement lors de l'appel du programme. A la place de la définition :

```
int main(void)
```

on peut écrire simplement : `main()`

2) Déclaration d'une fonction :

En C, il faut déclarer chaque fonction avant de pouvoir l'utiliser. La déclaration informe le compilateur du type des paramètres et du résultat de la fonction. A l'aide de ces données, le compilateur peut contrôler si le nombre et le type des paramètres d'une fonction sont corrects. Si dans le texte du programme la fonction est définie avant son premier appel, elle n'a pas besoin d'être déclarée.

Prototype d'une fonction :

La déclaration d'une fonction se fait par un ***prototype*** de la fonction qui indique uniquement le type des données transmises et reçues par la fonction.

Déclaration : Prototype d'une fonction

```
<Type> <NomFonct> (<TypePar1>, <TypePar2>, ...);
ou bien
<Type> <NomFonct> (<TypePar1> <NomPar1>, <TypePar2> <NomPar2>, ... );
```

IV) Renvoyer un résultat :

Par définition, toutes les fonctions fournissent un résultat d'un type que nous devons déclarer. Une fonction peut renvoyer une valeur d'un type simple ou l'adresse d'une variable ou d'un tableau.

Pour fournir un résultat en quittant une fonction, nous disposons de la commande **return** :

1) La commande return :

```
return <expression>;
```

L'instruction a les effets suivants :

- *évaluation de l'<expression>*
- *conversion automatique du résultat de l'expression dans le type de la fonction*
- *renvoi du résultat*
- *terminaison de la fonction*

Exemples :

La fonction CARRE du type **double** calcule et fournit comme résultat le carré d'un réel fourni comme paramètre.

```
double CARRE(double X)
{
    return X*X;
}
```

2) void :

En C, il n'existe pas de structure spéciale pour la définition de ***procédures*** comme en langage algorithmique. Nous pouvons cependant employer une fonction du type **void** partout où nous utiliserions une procédure en langage algorithmique.

Exemple :

La procédure LIGNE affiche L étoiles dans une ligne :

```
procédure LIGNE(L)
| donnée L
| (* Déclarations des variables locales *)
```

```
| entier I
| (* Traitements *)
| en I ranger 0
| tant que I<>L faire
| |     écrire "*"
| |     en I ranger I+1
| ftant (* I=L *)
| écrire (* passage à la ligne *)
fprocédure
```

Pour la traduction en C, nous utilisons une fonction du type **void** :

```
void LIGNE(int L)
{
/* Déclarations des variables locales */
int I;
/* Traitements */
for (I=0; I<L; I++)
   printf("*");
printf("\n");
}
```

3) exit :

exit nous permet d'interrompre l'exécution du programme en fournissant un code d'erreur à l'environnement. Pour pouvoir localiser l'erreur à l'intérieur du programme, il est avantageux d'afficher un message d'erreur qui indique la nature de l'erreur et la fonction dans laquelle elle s'est produite.

Une version plus solide de TAN se présenterait comme suit :

```
#include <math.h>

double TAN(double X)
{
   if (cos(X) != 0)
     return sin(X)/cos(X);

   else
    {
       printf("\aFonction TAN :\n Erreur : Division par zéro !\n");
       exit(-1); /* Code erreur -1 */
    }
}
```

V) Paramètres d'une fonction :

Les *paramètres* ou *arguments* sont les 'boîtes aux lettres' d'une fonction. Elles acceptent les données de l'extérieur et déterminent les actions et le résultat de la fonction. Techniquement, nous pouvons résumer le rôle des paramètres en C de la façon suivante :

*Les paramètres d'une fonction sont simplement des **variables locales** qui sont initialisées par les **valeurs** obtenues lors de l'appel.*

1) Passage des paramètres par valeur :

En C, le passage des paramètres se fait toujours par la valeur, c'est-à-dire les fonctions n'obtiennent que les ***valeurs*** de leurs paramètres et n'ont pas d'accès aux variables elles-mêmes.

Les paramètres d'une fonction sont à considérer comme des ***variables locales*** qui sont initialisées automatiquement par les valeurs indiquées lors d'un appel.

A l'intérieur de la fonction, nous pouvons donc changer les valeurs des paramètres sans influencer les valeurs originales dans les fonctions appelantes.

Avantages

Le passage par *valeur* a l'avantage que nous pouvons utiliser les paramètres comme des variables locales bien initialisées. De cette façon, nous avons besoin de moins de variables d'aide.

2) Passage de l'adresse d'une variable :

Comme nous l'avons constaté ci-dessus, une fonction n'obtient que les valeurs de ses paramètres. *Pour changer la valeur d'une variable de la fonction appelante,* nous allons procéder comme suit :

- la fonction appelante doit ***fournir l'adresse de la variable*** et la fonction appelée doit ***déclarer le paramètre comme pointeur***. On peut alors atteindre la variable à l'aide du pointeur.

Discussion d'un exemple :

Nous voulons écrire une fonction PERMUTER qui échange le contenu de deux variables du type **int**. En première approche, nous écrivons la fonction suivante :

```
void PERMUTER (int A, int B)
{
        int AIDE;
        AIDE = A;
        A=B;
        B = AIDE;
}
```

Nous appelons la fonction pour deux variables X et Y par :

```
PERMUTER(X, Y);
```

Résultat : X et Y restent inchangés !
Explication : Lors de l'appel, les *valeurs* de X et de Y sont copiées dans les paramètres A et B. PERMUTER échange bien contenu des variables *locales* A et B, mais les valeurs de X et Y restent les mêmes.

Pour pouvoir modifier le contenu de X et de Y, la fonction PERMUTER a besoin des adresses de X et Y. En utilisant des pointeurs, nous écrivons une deuxième fonction :

```
void PERMUTER (int *A, int *B)
{
        int AIDE;
        AIDE = *A;
        *A = *B;
        *B = AIDE;
}
```

Nous appelons la fonction par :

```
PERMUTER(&X, &Y);
```

Résultat : Le contenu des variables X et Y est échangé !
Explication : Lors de l'appel, les *adresses* de X et de Y sont copiées dans les *pointeurs* A et B. PERMUTER échange ensuite le contenu des adresses indiquées par les pointeurs A et B.

3) Passage de l'adresse d'un tableau à une dimension :

a) Déclaration :

Dans la liste des paramètres d'une fonction, on peut déclarer un tableau par le nom suivi de crochets,

```
<type> <nom>[]
```

ou simplement par un pointeur sur le type des éléments du tableau :

```
<type> *<nom>
```

Exemple :

La fonction **NBcarcter** calcule et retourne la longueur d'une chaîne de caractères fournie comme paramètre :

```
int NBcarcter (char *S)
{
   int N;
     for (N=0; *S != '\0'; S++)
         N++;
     return N;
}
```

b) Appel :

Lors d'un appel, l'adresse d'un tableau peut être donnée par le nom du tableau, par un pointeur ou par l'adresse d'un élément quelconque du tableau.

Exemple :

Après les instructions,

```
char CH[] = "Bonjour !";
char *P;
P=CH;
```

nous pouvons appeler la fonction **NBcarcter** définie ci-dessus par :

```
NBcarcter (CH)      /* résultat : 9 */
NBcarcter (P)       /* résultat : 9 */
NBcarcter (&CH[4])  /* résultat : 5 */
```

NBcarcter (P+2) **/* résultat : 7 */**
NBcarcter (CH+2) **/* résultat : 7 */**

Dans les trois derniers appels, nous voyons qu'il est possible de fournir *une partie* d'un tableau à une fonction, en utilisant l'adresse d'un élément à l'intérieur de tableau comme paramètre.

4) Passage de l'adresse d'un tableau à deux dimensions :

Exemple :

Imaginons que nous voulons écrire une fonction qui calcule la somme de tous les éléments d'une matrice de réels A dont nous fournissons les deux dimensions N et M comme paramètres.

Problème :

Comment pouvons-nous passer l'adresse de la matrice à la fonction ?

Par analogie avec ce que nous avons vu au chapitre précédent, nous pourrions envisager de déclarer le tableau concerné dans l'en-tête de la fonction sous la forme A[][]. Dans le cas d'un tableau à deux dimensions, cette méthode ne fournit pas assez de données, parce que le compilateur a besoin de la deuxième dimension du tableau pour déterminer l'adresse d'un élément A[i][j].

Une solution praticable consiste à faire en sorte que la fonction reçoive un pointeur (de type float*) sur le début de la matrice et de parcourir tous les éléments comme s'il s'agissait d'un tableau à une dimension N*M.

Cela nous conduit à cette fonction :

```
float SOMME(float *A, int N, int M)
{
int I;
float S;
  for (I=0; I<N*M; I++)
     S += A[I];
  return S;
}
```

Lors d'un appel de cette fonction, la seule difficulté consiste à transmettre l'adresse du début du tableau sous forme d'un pointeur sur **float**. Prenons par exemple un tableau déclaré par

```
float A[3][4];
```

Le nom A correspond à la bonne adresse, mais cette adresse est du type *"pointeur sur un tableau de 4 éléments du type **float**"*. Si notre fonction est correctement déclarée, le compilateur la convertira automatiquement dans une adresse du type *'pointeur sur **float**'*.

Solution :

Voici finalement un programme faisant appel à notre fonction SOMME :

```
#include <stdio.h>
main()
{/* Prototype de la fonction SOMME */
float SOMME(float *A, int N, int M);

/* Déclaration de la matrice */
float T[3][4] = {{1, 2, 3, 4},{5, 6, 7, 8},{9,10,11,12}};

/* Appel de la fonction SOMME */
printf("Somme des éléments : %f \n",SOMME((float*)T, 3, 4) );
return (0);
}
```