Chapitre 4

Les fonctions

Chapitre 4 : Les fonctions

I Introduction

- Le langage **C** intègre le concept de programmation modulaire.
- Lorsqu'un programme a une taille conséquente, il est nécessaire et utile de le découper en sous-programmes ⇒ ***fonctions***
- La notion de ***fonction*** permet d'éviter la duplication de morceaux de code identiques et simplifie la mise au point du programme.
- En **C**, il n'existe qu'un seul type de ***fonction*** qui remplace les notions de fonction et module vus en algorithmique.
- *La notion de module telle que vue en algorithmique n'existe pas à proprement parler en C.*

Chapitre 4 : Les fonctions

II Généralités

- Une **fonction** est définie **avant** le programme principal (***main***).
- Elle peut être définie **après** à condition de spécifier son **profil** (ou **entête**) **avant**.
- Le programme principal ***main*** est une **fonction**. Elle doit exister obligatoirement et peut appeler une ou plusieurs *fonctions secondaires*.
- Chacune de ces *fonctions* peut appeler d'autres *fonctions*.
- Une **fonction** possède un **type** : Il correspond au type du résultat renvoyé.
- Une **fonction** qui n'a pas de **type** (*ne renvoie pas de résultat explicitement*) est également appelée ***procédure***. On peut toutefois lui spécifier un type avec le mot-clé ***void*** (qui signifie littéralement *vide*).
- Par convention, le début d'exécution d'un programme **C** est donné à la fonction principale ***main***.

Chapitre 4 : Les fonctions

III La fonction

a Définition de la fonction

Elle est composée d'un *entête* (ou prototype) et d'un *corps*.

Le *corps* comprend des **déclarations** et des **instructions**.

Syntaxe :

```
type_fonction  nom_fonction ( [liste_paramètres_formels] )   /*prototype */
{
        [liste_déclarations_locales]          /* optionnelle*/

        liste_instructions  /*corps */
}
```

Chapitre 4 : Les fonctions

✓ *type_fonction* : type du résultat

✓ *liste_paramètres_formels* :

- **type_paramètre_1** nom_paramètre_1**, ...**, **type_paramètre_n** nom_paramètre_n
- Cette liste peut être vide : c'est le cas d'une fonction sans paramètres

✓ *liste_déclarations* : déclarations de variables locales à la fonction, inaccessibles à l'extérieur de la fonction.

- Leur portée est limitée à la fonction.

✓ *liste_instructions* : ensemble d'instructions exécutées à l'appel de la fonction. Parmi cet ensemble, une instruction spécifique ***return*** qui permet de renvoyer le résultat de la fonction.

- *Utilisation* : **return** expression**;**

Chapitre 4 : Les fonctions

Remarques :

❑On peut avoir plusieurs instructions **return** dans une fonction. Le retour à l'appelant (*fonction principale* ou *fonction secondaire*) est provoqué par la première instruction **return** rencontrée à l'exécution.

❑Dans la définition de l'**en-tête**, le *type de la fonction* peut être optionnel. Si le programmeur l'omet, le compilateur rend par défaut le type *int*.

❑*Attention : éviter de prendre cette mauvaise habitude de programmation dans le cas d'une fonction qui doit rendre un int.*

Chapitre 4 : Les fonctions

Exemples :

1)

```
int  Max2 ( int a, int b )   /* la fonction renvoie un entier */
{
   int res ;                 /* variable locale */
   if ( a > b )  res = a ;
   else res = b ;
   return  res ;             /* retour à l'appelant avec le résultat */
}
```

2)

```
int  Somme_Carres (int i, int j)  /* la fonction renvoie un entier */
{
   int res ;                 /* variable locale */
   res = i*i + j*j ;
   return res ;              /* retour à l'appelant avec le résultat */
}
```

3)

```
double Pi ()                 /* pas de paramètres formels */
{                            /* pas de variables locales */
   return ( 3.14159 ) ;      /* l'expression peut être entre ( ) */
}
```

4)

```
void  Affiche ( float prix )
{
printf ( " Prix affiché = %3.2f \n ", prix ) ;
}
```

Remarque : une fonction avec un type void est l'équivalent d'un module (tel que vu en algorithmique)

Chapitre 4 : Les fonctions

- ✓ Une même fonction peut avoir différentes écritures.
- ✓ La fonction Max2 qui calcule le maximum entre deux entiers peut être écrite :

```
int Max2 ( int a, int b )
{
 if  ( a > b )  return a ;
 else return b ;
}
```

Ou :

```
int Max2 ( int a, int b )
{
 int res ;
 res = ( a > b ? a : b ) ;
 return res ;
}
```

Ou :

```
int Max2 ( int a, int b )
{
return ( a > b ? a : b ) ;
}
```

Chapitre 4 : Les fonctions

Autre exemple : La fonction Max3 qui calcule le maximum entre trois entiers peut être écrite :

```
int Max3 ( int  a, int  b, int  c )
{
  int max ;
  max = Max2 ( a , b ) ;
  max = Max2 ( max , c ) ;
  return max ;
}
```

Ou :

```
int Max3 ( int  a, int  b, int  c )
{
  int max ;
  max = Max2 ( a , Max2 ( b , c ) ) ;
  return max ;
}
```

Ou :

```
int Max3 ( int a, int b, int  c )
{
return Max2 ( a , Max2 ( b , c ) ) ;
}
```


Chapitre 4 : Les fonctions

b Appel à la fonction

Syntaxe : nom_fonction **(***paramètre_effectif_1***, ……, ***paramètre_effectif_n***)**

- *les paramètres effectifs peuvent être des expressions*

Rappel : *Un appel de fonction avec un type autre que **void** (qui n'est pas une procédure) est remplacé par la valeur calculée.*

Ce n'est pas une instruction.

Chapitre 4 : Les fonctions

Exemple :

Version 1

```
#include <stdio.h>
int Max2 ( int a, int b )
{
return ( a > b ? a : b ) ;
}
int Max3 ( int a, int b, int c )
{
return Max2 ( a, Max2 ( b , c ) ) ;
}
void Affiche (int a, int b, int c, int m )
{
printf ( " Max entre %d, %d, %d = %d ", a, b, c , m);
}
```

```
int main ( )
{
 int val1, val2, val3 ;

 printf ( " Saisir 3 entiers quelconques : \n " ) ;
 scanf ( " %d %d %d " , &val1 , &val2 , &val3 ) ;
 Affiche ( val1, val2, val3, Max3 ( val1 , val2 , val3 ) ) ;
}
```

Si les valeurs saisies sont 4, 7 et -3 le résultat affiché est :

Max entre 4, 7 , -3 = 7

Chapitre 4 : Les fonctions

Version 2

```
#include <stdio.h>
int Max2 ( int a, int b ) ;
int Max3 ( int a, int b, int c ) ;
void Affiche (int a, int b, int c, int m ) ;
int main ( )
{
 int val1, val2, val3 ;

 printf ( " Saisir 3 entiers quelconques : \n " ) ;
 scanf ( " %d %d %d " , &val1 , &val2 , &val3 ) ;
 Affiche ( val1, val2, val3, Max3 ( val1, val2, val3 ) ) ;
}
```

```
int Max2 ( int a, int b )
{
return ( a > b ? a : b ) ;
}
int Max3 ( int a, int b, int c )
{
return Max2 ( a, Max2 ( b , c ) ) ;
}
void Affiche (int a, int b, int c, int m )
{
printf ( " Max entre %d, %d, %d = %d ", a, b, c , m) ;
}
```

Chapitre 4 : Les fonctions

IV La fonction sans « retour » ou procédure

Pour exprimer cette notion de fonction ne renvoyant aucune valeur :

✓ on associe un type spécial du langage, le type ***void***, à l'entête de la fonction

✓ on n'utilise pas d'instruction ***return*** dans la partie *liste_instructions* du *corps* de la fonction.

Remarques :

➢ On utilisera la désignation ***fonction*** pour désigner indifféremment une **procédure** ou une **fonction** proprement dite à chaque fois qu'il ne sera pas nécessaire de faire la distinction entre les deux.

➢ Selon le compilateur, le type **void** peut être omis.

➢ On peut tout de même utiliser l'instruction ***return*** toute seule, sans *expression* associée :

return ;

Ce qui a même effet que de ne pas l'utiliser.

Chapitre 4 : Les fonctions

Exemple :

Version 1

```
#include  <stdio.h>
void Affich_add ( int  i, int  j )                    /* somme et affichage */
{
  int res = i + j ;                                   /* déclaration + initialisation */
  printf (" Somme de %d et %d  = %d ", i, j, res) ; /* affichage du résultat */
}
void  Affiche ()
{
   int a = 4, b = 8 ;
 /* appels de Affich_add */
 Affich_add ( a, b ) ;
 Affich_add ( 12, b ) ;
}
int  main ()
{
…….
Affiche () ;
……..
}
```

Version 2

```
#include  <stdio.h>
void Affich_add ( int  i, int  j ) ;
void  Affiche () ;
int  main ()
{
…….
Affiche () ;
……..
}
void Affich_add ( int  i, int  j )
{
  int res = i + j ;                                   /* déclaration + initialisation */
  printf (" Somme de %d et %d  = %d ", i, j, res) ; /* affichage du résultat */
}
void  Affiche ()
{
  int a = 4, b = 8 ;
 /* appels de Affich_add */
 Affich_add ( a, b ) ;
 Affich_add ( 12, b ) ;
}
```

Chapitre 4 : Les fonctions

V Transmission des paramètres

- En **C**, le mode de passage des paramètres est différent de ce qui a été vu en algo et de ce qui existe dans d'autres langages de programmation.
- La transmission des paramètres se fait par valeurs : il y a re-copie des valeurs des *paramètres effectifs* dans les *paramètres formels*.
- La **fonction** travaille uniquement sur la copie.
- Si la **fonction** est amenée à modifier le paramètre formel, seule la copie est modifiée car les *paramètres formels* sont traités de la même manière que les *variables locales* de la fonction.
- La modification n'a aucune incidence sur le paramètre effectif de l'appelant.
- Le passage par valeur correspond au *mode entrée* (↓)

Chapitre 4 : Les fonctions

Exemple :

```
# include  <stdio.h>
void Permut ( int  a, int  b )
{
  int mem = a ;
  a = b ;
  b = mem ;
}
```

```
int main ()
{
  int x = 3, y = 6;
  Permut ( x, y ) ;
  printf ( " x = %d, y = %d \n", x, y ) ;
}
```

Résultat affiché :

x = 3, y = 6

Chapitre 4 : Les fonctions

- Pour **modifier** le contenu des deux *paramètres effectifs*, il faut donner à la fonction **Permut** leurs ***adresses*** (*adresse-mémoire*) et non pas leurs *valeurs* (*contenu de l'emplacement-mémoire*)
- Cela doit être fait **explicitement** par le programmeur
- *Le passage de paramètres par **adresse** correspond à la notion de paramètres d'entrée-sortie et de sortie telle que vue en algo.*

Chapitre 4 : Les fonctions

Exemple :

```
#include <stdio.h>
void Permut ( int *a, int *b )
{
  int mem = *a ;
 *a = *b ;
 *b = mem ;
}
```

a,b : variables de type pointeur, contiennent les @ d'emplacements mémoires de type entier

**a, *b : contenu de ces emplacements mémoires*

```
int main ( )
{
  int x = 3, y = 6;
  Permut ( &x, &y ) ;
  printf ( " x = %d, y = %d \n", x, y ) ;
}
```

Résultat affiché :

x = 6, y = 3

x, y : variables de type entier

&x, &y : adresses-mémoire de ces variables