# Partie III : Procédures et fonctions

## LES SOUS PROGRAMMES

- Un programme écrit en langage C ou autre langage de programmation devient de plus en plus important, c'est-à-dire le nombre de lignes de code est très grand. A cet effet, il est conseiller de partager ce programme en plusieurs sous programmes (modules) et chaque module aura une tache bien déterminée à traiter. On parle d'une analyse descendante.
- On distingue deux (02) genres de sous programmes : les procédures et les fonctions (voir cours algorithmique) mais le langage C permet de définir seulement les fonctions.
- En effet une procédure en langage C n'est autre qu'une fonction qui ne retourne aucun résultat dans son nom !

## POURQUOI LES SOUS PROGRAMMES ?

- L'utilisation des sous programmes est plus que nécessaire car elle nous permet de raffiner notre programme et arriver à avoir des modules élémentaires faisant une et une seule tache.
- Un programme utilisant des sous programmes est lisible donc facile à comprendre.
- En résumé les sous programmes sont utilisés pour les raisons suivantes :
  - Raffinage des programmes donc lisibilité des programmes.
  - Réutilisation des modules.
  - Compilation des modules à part donc la maintenance est beaucoup plus facile.
  - Gain de temps (CPU).
  - Gain d'espace mémoire (MC).
  - …

## DEFINITION D'UNE FONCTION

- Une fonction est un sous-programme qui a une tache unique, peut être utilisée autant de fois que l'on veut (réutilisabilité) et peut retourner une valeur à l'appelant (le programme ayant fait appel à cette fonction).
- En langage C, on ne parle seulement de fonction à moins qu'on utilise une fonction de type void. C'est une fonction qui ne retourne aucune valeur : elle correspond au concept de procédure. Le mot clé void permet de déclarer un objet sans type. Un exemple sera donné par la suite pour illustrer ce genre de fonction.

- En général, l'écriture des fonctions se fait après l'entête et avant main() qui représente à son tour une fonction principale (Par abus de langage on dit que c'est le programme principal).
- A vrai dire on ne parle pas d'écriture d'une fonction mais plutôt d'une déclaration de fonction vu l'endroit qu'elle occupe dans le programme.

## COMMENT DECLARER UNE FONCTION ?

- Une fonction est déclarée par son type, son nom et ses arguments qu'on appelle aussi paramètres.
- La syntaxe de déclaration est la suivante :

  Type nom_de_la_fonction (typ1 arg1, typ2 arg2, … , typn argn)
- Une fois la fonction est déclarée, on définit son corps qui sera composé éventuellement de déclarations de variables (variables locales propres à cette fonction) et des instructions.

  Voici la syntaxe :

```
Type nom_de_la_fonction (typ1 arg1, typ2 arg2, … , typn argn)
{
        Déclaration_de_variables ;
        Instructions ;
}
```

Parmi les instructions, il y a l'instruction **return** qui fait retourner le résultat à l'appelant.

**Exemple :** Le maximum entre deux nombres entiers.

```
int  max(Int a, Int b)
{
        If  (a > b) return a ;
        return b;
}
```

L'instruction return casse la séquence d'instructions, c'est à dire elle nous permet de sortir de la fonction sans se soucier de ce qu'il vient par la suite.

Voici une autre écriture de l'exemple précédant proche de l'algorithmique :

```
int  max(Int a, Int b)
{
        If  (a > b)
                Return a ;

        else

                return b;
}
```

- Du moment qu'on ait déclaré et défini une fonction, on l'utilise dans le programme principal main() comme s'il s'agit d'une fonction intégrée au langage C. On l'appelle par son nom ou on l'affecte à une variable de même type.

**Exemple de bilan :**

```
/* Première écriture appel par le nom*/
#include <stdio.h>
int  max(Int a, Int b)
{
        if  (a > b)
                return a ;
        else
                return b;
}
main()
{
        int x;
        int y;
        scanf(''%d'', &x);
        scanf(''%d'', &y);
        printf(''%d'', max(x, y));        /*On voit bien l'appel par le nom*/
}
```

```
/* Deuxième  écriture appel par l'affectation*/
#include <stdio.h>
int  max(Int a, Int b)
{
        if  (a > b)

                return a ;

        else

                return b;
}
main()
{
        int x;
        int y;
        int m ;
        scanf(''%d'', &x);
        scanf(''%d'', &y);
        m = max(x, y) ;                 /*l'appel est par l'affectation*/
        printf(''%d'', m);
}
```

## « PROCEDURE » EN LANGAGE C

- Voici maintenant comment on définit une « procédure » en langage C : on met le mot clé void pour dire qu'il n'y ait pas de résultat.
  **Exemple :** on écrit une procédure qui affiche seulement un nombre entier.

```
void afficher(int nb)
{
        Printf(''nb= %i\n'', nb) ;
}
```

- Utilisant cette procédure dans le programme écrit ci-dessus :

  C'est-à-dire on aura une fonction principale main(), une fonction max() et la procédure afficher.

```
/* Nouveau programme*/
#include <stdio.h>
int  max(int a, int b)            /* la fonction max()*/
{
        if  (a > b)

                return a ;

        else

                return b;
}

void afficher(int nb)             /*la procedure afficher*/
{
        printf(''nb= %i\n'', nb) ;
}

main()                            /*la fonction principale*/
{
        int x;
        int y;
        scanf(''%d'', &x);
        scanf(''%d'', &y);
        afficher(max(x, y));
}
```

**NB:** On remarque que dans la procédure (void afficher(int nb)) il n'y a pas de return (return sans valeur).

- Un autre exemple de procédure qui nous permet de sauter x lignes.

```
void  sauter_ligne(int x)
{
        int co ;
        for (co = 1 ; co <= x ; co = co+1;)
                printf(''\n'');
}
```

C'est comme si on écrit l'instruction printf(''\n''), x fois!

## VARIABLES GLOBALES ET VARIABLES LOCALES

- Une variable globale est déclarée en général dans le programme principal mais en langage C, elle est déclarée en dehors de toutes les fonctions même la fonction principale main() qui est considérée comme étant le programme principal. Une fois la variable globale est déclarée, elle sera utilisable par toutes les fonctions définies dans le programme.
- Une variable locale est déclarée à l'intérieur d'une fonction et elle ne peut être utilisée que par celle-ci !
- Exemple de bilan sur les variables globales et les variables locales.

```
#include <stdio.h>
Int x, y ;                      /*x et y sont des variables globales*/
void echanger(int A, int B)
{
        int  C ;                /* C  est une variable locale*/
        A = x ;
        B = Y ;
        C = A ;
        A = B ;
        B = C ;
        x  = A ;
        y = B ;
}

main()

{

        printf(''Entrer x et y \n'') ;

        scanf(''%i'', &x) ;     /*on a utilisé x et y et pourtant ils ne sont pas*/

        scanf(''%i'', &y) ;     /* déclarés dans la fonction principale main()*/

        echange(x, y) ;         /* car ils sont des variables globales*/

        printf(''x = %i et y = %i'' , x, y) ;

}
```

**NB :** Il est conseillé de ne pas utiliser les variables globales car on ne sera jamais exempt de pas mal de surprises ! (Effet de bord).

## PARAMETRES

- Regardons l'exemple ci-dessus, on a dit que x et y sont des variables globales, C est une variable locale et qu'appelle-t-on A et B qu'on voit dans la fonction void echanger(Int A, Int B) ?
  A et B sont appelés paramètres ou arguments.
- On distingue deux (02) genres de paramètres : les paramètres formels et les paramètres réels.
  - Quant aux paramètres formels, ce sont ceux qu'on voit dans la déclaration d'une fonction. Exemple void echanger(Int A, Int B). Donc on peut dire que A et B sont des paramètres formels.
    Les paramètres formels n'ont pas d'existance en mémoire centrale (MC).
  - Quant aux paramètres réels, appelés aussi paramètres d'appel, on les trouve au moment de l'appel de la fonction, comme par exemple echanger(x, y) ; donc x et y sont des paramètres réels.

**NB :** Attention les paramètres formels d'une fonction doivent être au même nombre, dans le même ordre et de même type que les paramètres d'appel.

## PASSAGES DE PARAMETRE

- Avant de parler des modes de passage des paramètres, il est impératif de donner quelques notions sur l'adressage ou pointeur.
  En langage C, un pointeur est une variable contenant l'adresse d'une autre variable d'un type donné. Sa déclaration se fait comme suit :

  ```
  type   *nom_du_pointeur ;
  ```

  **Exemple :** Int *pt ;
  Dans ce cas, on a déclaré un pointeur appelé pt qui va contenir l'adresse d'une variable de type entier.
  Si on déclare une variable X de type entier par Int X ; On peut se demander comment aurait-on l'adresse de cette variable X ?
  C'est très simple il suffit de précéder cette dernière par le caractère &, donc &X est l'adresse de la variable X.
  Si on écrit l'instruction pt= &X ; on dit qu'on a affecté l'adresse de la variable X au pointeur pt.
  Donc pt est un pointeur qui contient l'adresse de la variable X de type entier.

- **Exemple de bilan.**

```
#include <stdio.h>
main()
{
        int X ;                 /*Déclaration de la variable X de type entier*/
        int  *pt ;              /*Déclaration du pointeur pt*/
        printf(''Entrer une valeur entière'') ;
        scanf(''%d'', X) ;      /*Lecture d'une valeur dans X*/
        pt = &X ;               /* pt contient l'adresse de la variable X*/
}
```

- On va étudier deux modes de passages de paramètres : l'un par valeur et l'autre par variable.
  - Le passage par valeur : Dans ce cas on dit qu'il y ait lecture mais pas d'écriture ; c'est-à-dire au moment de l'appel les paramètres réels sont affectés l'un après l'autre dans les paramètres formels (on dit qu'il y a lecture) mais au moment du retour à l'appelant rien n'est fait (on dit qu'il n'y a pas d'écriture) et par conséquent il n'y a pas de modifications des paramètres réels.
  - Le passage par variable, il est appelé aussi passage par adresse : Dans ce cas on dit qu'il ait lecture et écriture ; cela veut dire qu'au moment de l'appel les paramètres réels sont affectés l'un après l'autre dans les paramètres formels (on dit qu'il y a lecture), après le traitement au sein de la fonction et au moment du retour à l'appelant les informations qui sont dans les paramètres formels sont affectés dans les paramètres réels (on dit qu'il y a écriture) . Par conséquent les valeurs des paramètres réels seront modifiées.

**NB :** Ces explications sont plutôt schématiques, à vrai dire si le passage est par valeur donc la fonction appelée traite l'information et ne modifie en aucun cas la variable passée par l'appelant ; par contre si le passage est par variable, la fonction appelée reçoit l'adresse de la valeur et non pas la valeur elle-même. A l'aide de cette adresse la machine va pointer l'emplacement physique et traite l'information qui y est. Donc automatiquement cette information va éventuellement changer !

- Exemple de bilan sur le passage par valeur.

```
/* l'appel est par valeur*/
#include <stdio.h>
void echanger(Int V1, Int V2)
{
        int inter ;
        inter = V1 ;            /*on est en train de manipuler*/
        V1 = V2 ;               /*les valeurs*/
        V2 = inter ;
}
main()
{
        int a, b ;
        a = 14 ;
        b = 23 ;
        printf(''avant : a = %d  b = %d\n'', a,b) ;
        echanger(a, b) ;        /*Envoi des valeurs de a et de b*/
        printf(''aprés : a = %d  b = %d\n'', a,b) ;
}
```

**Après le traitement le résultat est :**

```
Avant : a = 14  b = 23
Après : a = 14  b = 23
```

On remarque que malgré l'échange qui a été fait à l'intérieur de la fonction echanger(), a et b sont restées telles qu'elles sont.

- Exemple de bilan sur le passage par variable.

```
/* l'appel est par variable*/
#include <stdio.h>
void echanger(int *V1, int *V2)
{
        int inter ;
        inter = (*V1) ;                 /*On est en train de manipuler*/
        (*V1) = (*V2) ;                 /*les adresses*/
        (*V2) = inter ;
}
main()
{
        int a, b ;
        a = 14 ;
        b = 23 ;
        printf(''avant : a = %d  b = %d\n'', a,b) ;
        echanger(&a, &b) ;              /*Envoi des adresses de a et de b*/
        printf(''aprés : a = %d  b = %d\n'', a,b) ;
}
```

**Après le traitement le résultat est :**

Avant : a = 14  b = 23
Après : a = 23  b = 14

On remarque que les valeurs des variables a et b ont changées, donc il y a bel et bien un échange entre a et b !

## UTILISATION DES FONCTIONS MATHEMATIQUES

- Avant l'utilisation des fonctions mathématiques qui existent en langages C, il faut les inclure pour que le compilateur C les reconnaisse. Cette inclusion est provoquée par la directive #include.

  #include <math.h>

- Une fois elles sont incluses, on peut utiliser les fonctions mathématiques telles que racines, logarithmes, trigonométrie, …
- Voici quelques fonctions :

| | |
|---|---|
| Sqrt(a) | racine carrée de a |
| Pow(a, b) | $a^b$ |
| Log(a) | logarithme népérien de a |
| Log10(a) | logarithme de base 10 de A |
| Exp(a) | $e^a$ |
| Sin(a) acos(a) tanh(a) … | fonctions trigonométriques et hyperboliques |
| … | |

**NB :** Une petite mise au point concernant le module « main ». On voit souvent l'entête de main s'écrit comme fonction de type entier et avec deux paramètres : argv et argc.

```
Int main(Int argc, char *argv[ ])
{
    .
    .
    .
    Retun 0 ;
}
```

C’est quoi ces paramètres argv et argc ?

- Argv est un tableau (vecteur) de pointeurs et chacun de ces pointeurs pointe une chaîne de caractères.
- Argc indique simplement le nombre de chaînes de caractères sur lequel pointe argv.

A quoi cela va-t-il servir ?

En réalité c’est le système d’exploitation (windows dans notre cas) qui envoie ces chaînes de caractères au programme au moment du lancement de ce dernier.

La première chaîne de caractères indique en général le chemin du programme.

Par exemple : C:/programme /calculer.exe

Cela veut dire que le programme exécutable appelé « calculer.exe » se trouve dans la directory (dossier) appelée « programme » et cette dernière est dans le disque dur (C).

# ANNEXES

**Tableau récapitulatif des types de données en langage C**

| Types de données | Signification | Taille en octets | Plage de valeurs acceptées |
|---|---|---|---|
| char | Caractère | 01 | -128 à 127 |
| Unsigned char | Caractère non signé | 01 | 0 à 255 |
| Short Int | Entier court | 02 | -32768 à 32767 |
| Unsigned short Int | Entier court non signé | 02 | 0 à 65535 |
| Int | Entier | 02 sur processeur 16 bits<br>04 sur processeur 32 bits | -32768 à 32767<br>-2147 483 647 à 2147 483 647 |
| Unsigned Int | Entier non signé | 02 sur processeur 16 bits<br>04 sur processeur 32 bits | 0 à 65535<br>0 à 4294 967 295 |
| Long Int | Entier long | 04 | -2147 483 647 à 2147 483 647 |
| Unsigned long Int | Entier long non signé | 02 | 0 à 4294 967 295 |
| float | Réel | 04 | $3.4*10^{-38}$ à $3.4*10^{38}$ |
| Double | Réel double | 08 | $1.7*10^{-308}$ à $1.7*10^{308}$ |
| Long double | Réel long double | 10 | $3.4*10^{-4932}$ à $3.4*10^{4932}$ |

**Tableau récapitulatif des indicateurs de format en langage C**

| Indicateurs | Descriptions |
|---|---|
| %c | Format caractère |
| %d | Format entier |
| %i | Format entier (équivalent à %d) |
| %f | Format réel |
| %e ou %E | Format de notation scientifique |
| %g ou %G | Utilise %f ou %e selon la longueur de la valeur décimale |
| %o | Format octal non signé |
| %s | Format chaîne de caractères |
| %u | Format entier non signé |
| %x | Format hexadécimal non signé (Les lettres en minuscules) |
| %X | Format hexadécimal non signé (Les lettres en majuscules) |
| %p | Affichage du pointeur d'argument |
| %n | Enregistre le nombre de caractères affichés |
| %% | Permet d'afficher le caractère % |

**Tableau des caractères d'échappement**

| Caractères | Descriptions |
|---|---|
| \n | Saut de ligne (retour chariot) |
| \t | Caractère tabulation |
| \b | Déplace le curseur d'un caractère vers la gauche |
| \r | Place le curseur au début de la ligne en cours |
| \f | Saut de page (place le curseur au début de la page suivante) |

## BIBLIOGRAPHIE

1- Langage C : Cours et références
Pierre NERZIC Mars 2003

2- Langage C : Support de cours
Patrick CORDE Mars 2006

3- Programmation en langage C
M.C. BELAID

4- Algorithmique, structures de données et langage C
J.M. ENJALBERT Janvier 2005