# Chapitre 06 : Les sous-programmes

## 1. Introduction :

Dans la résolution d'un problème, on peut constater qu'une suite d'actions revient plusieurs fois. Dans ce cas il serait judicieux de l'écrire une seule fois, et de l'utiliser autant de fois que c'est nécessaire. Par exemple, pour calculer le nombre de combinaison k parmi N

$$C_n^k = \frac{n\,!}{k\,!\,(n-k)\,!}$$

Il faut calculer la factoriel d'un entier trois fois pour des valeurs différentes. Il est judicieux d'écrire un sous-programme qui calcule la factoriel d'un entier et de l'utiliser autant de fois que nécessaire.

Lorsque le problème à résoudre est très complexe, il est préférable de le découper en plusieurs modules et développer un sous-programme pour chaque module (programmation modulaire).

Un algorithme utilisant des sous-programmes s'appelle l'algorithme principal. Nous distinguons en algorithmique deux types de sous-programme : les procédures et les fonctions.

## 2. Les procédures :

Une procédure est un sous-programme destiné pour réaliser un traitement particulier sur des données d'entrés afin de produire des résultats.

### a) Déclaration & appel :

Pour déclarer une procédure dans un algorithme, on suit la syntaxe suivante :

```
Procedure nom_procedure (P1 :type, P2 :type, …);
    Déclarations des variables locales ;
Debut
    Instructions;
Fin ;
```

Pour appeler la procédure dans un algorithme il suffit d'utiliser le nom de la procédure en indiquant la liste des paramètres requis. On écrit :

```
nom_procedure (P1,P2, …);
```

**Support de cours**

## b) Déclaration d'une procédure dans un algorithme :

```
Algorithme nom_algorithme ;
 Déclaration des variables ;
 Procedure nom_procedure (liste des paramètres)
     Declarations des variables locales ;
Debut
     Instructions de la procédure ;
Fin ;
Debut
     Instruction_1;
     Instruction_2;
     nom_procedure (liste des paramètres);
     Instruction_3;
 …
Fin.
```

**Exemple** : un algorithme qui affiche le maximum de deux nombres en utilisant une procédure.

```
Algorithme maximum;
Var x,y : reel ;
Procedure     afficher_max(a     :
reel,b :reel)
Var max : reel ;
Debut
     Si(a>b) alors
       max ← a;
       Sinon
         max ← b;
       Finsi ;
       Ecrire(max);
Fin;
Debut
     Lire(x,y);
     afficher_max(x,y);
Fin.
```

## c) La visibilité des variables :

Les variables déclarées dans le programme principal sont dites **globales**. Les variables globales sont visibles par le programme principal et tous les sous-programmes de l'algorithme. Les variables déclarées dans un sous-programme sont dites **locales** au sous-programme. Elles ne sont visibles que par le sous-programme qu'il les a déclarées.

Dans l'exemple précèdent, les variables `x` et `y` sont globales. La variable `max` est locale à la procédure `afficher_max`.

**Support de cours**

## d) Notion de paramètres :

Les paramètres servent de moyen de communication entre le programme appelant et le sous-programme appelé. Les paramètres définis dans l'entête de la procédure sont appelés des paramètres **fictifs.** Les paramètres transmis lors de l'appel d'une procédure sont dits **effectifs**. On distingue deux modes de passage de paramètres :

**Passage par valeur :**

C'est le mode de passage par défaut, il permet au programme appelant de transmettre une constante ou la valeur d'une variable à la procédure appelée. Dans ce cas les paramètres fictifs de la procédure sont considérer comme des variables locales à la procédure.

**Passage par adresse :**

Un paramètre d'une procédure utilisant le passage par adresse est précédé par le mot clés **var**. Le passage par adresse permet au programme appelant de transmettre l'adresse d'une variable au programme appelé. Lors de l'appel d'une procédure, les paramètres effectifs ayant un passage par adresse doivent être obligatoirement des variables. Le programme appelant envoie l'adresse de la variable passée en paramètre à la procédure. Dans ce cas, la procédure connait l'adresse mémoire de la variable reçue en paramètre et elle pourra donc la modifier.

**Remarque** :

- Une procédure admet des paramètres d'entrées et des paramètres de sortie.
- Les paramètres d'entrées utilisent le passage par valeur.
- Les paramètres de sortie utilisent le passage par adresse.
- Un paramètre peut être d'entré et de sortie en même temps, dans tel cas on utilise passage par adresse.

**Exemples :** Un algorithme qui calcule la somme $S(n) = 1 + 2 + 3 + \cdots + n$ en utilisant une procédure.

```
Algorithme calcul_somme;
Var s,n: entier;
Procedure somme(n:entier, var s:entier) ;
Debut
       s ← 0 ;
     Tantque(n<>0)faire
       s ← s +n ;
       n ← n-1 ;
     fintQ ;
Fin ;
```

```
Debut
     Lire(n);
     Somme(n,s) ;
     Ecrire(s) ;
Fin.
```

**Support de cours**

# 3. Les fonctions :

Une fonction est un sous-programme ayant un et un seul résultat. Une fonction renvoie toujours un résultat, c'est pour cette raison que nous devant préciser le type d'une fonction au moment de sa déclaration.

## a) Déclaration & appel :

La syntaxe algorithmique pour déclarer une fonction en algorithmique est la suivante :

```
fonction nom_fonction (P1 :type, P2 :type, …): type ;
     Déclarations des variables locales ;
Debut
     Instructions;
     Retourner(resultat);
Fin ;
```

Où type est le type du résultat retourné par la fonction. Le mot clés **`retourner`** est obligatoire dans une fonction car, une fonction doit toujours renvoyer un résultat. Le mot clés retourner provoque la fin d'exécution de la fonction.

Une fonction est appelée avec sont nom on indiquant ses paramètres comme suit :

```
nom_fonction (P1:type, P2 :type, …) ;
```

À l'exécution l'appel de la fonction est remplacé par la valeur retournée. L'appel de la fonction peut être affecté à une variable ou utilisé dans une expression.

**Exemple** : un algorithme qui calcule $C_n^k = \frac{n!}{k!(n-k)!}$ avec une fonction `Fact` qui calcule la factorielle d'un entier.

```
Algorithme calcul_somme;
Var k,n,C: entier;
fonction Fact(n:entier):entier;
var F:entier;
Debut
       F ← 1 ;
     Tantque (n<>0)faire
       F ← F * n ;
       n ← n-1 ;
     fintQ ;
     retourner(F);
Fin ;
```

```
Debut
     Lire(k,n);
     C ← Fact(n)/(Fact(k)*Fact(n-k));
     Ecrire(C) ;
Fin.
```

## b) Paramètre d'une fonction :

Du fait que tous les paramètres d'une fonction sont des paramètres d'entrés, on ne peut utiliser que le passage par valeurs. La seule sortie d'une fonction est le résultat renvoyé automatiquement.

# 4. La récursivité :

Un problème est dit récursif lorsque sa résolution dépond de la solution de lui-même à un ordre inferieur. Voici quelques exemples de calculs récursifs :

- Factorielle (n) = n * n-1 * … * 2 * 1 = n * Factorielle (n-1)
- S (n) = n + n-1 + … + 2+1 = n + S (n-1)
- $X^n = n * X^{n-1}$

En algorithmique on peut réaliser un calcul récursif en utilisant un sous-programme récursif. C.à.d. pour réaliser le calcul à l'ordre n le sous-programme va utiliser le résultat du même calcul à un ordre inférieur. Pour cela le sous-programme va appeler lui-même.

Pour qu'un sous-programme récursif s'arrête d'appeler lui-même, le résultat du calcul récursif doit être évident pour un certain ordre. Ci-dessous les relations de récursivité des calculs cités précédemment avec leurs termes permettant l'arrêt de l'appel récursif :

- $factorielle(n) = \begin{cases} 1, si\ n = 0 \\ n * factorielle(n-1),\ sinon \end{cases}$
- $S(n) = \begin{cases} 0, si\ n = 0 \\ n + S(n-1),\ sinon \end{cases}$
- $X^n = \begin{cases} 1, si\ n = 0 \\ X * X^{n-1},\ sinon \end{cases}$

**Exemples :** la fonction récursive qui calcule la factorielle d'un entier.

```
fonction Fact(n:entier):entier;
Debut
     Si(n=0)Alors
       Retourner(1) ;
      Sinon
       Retourner(n*Fact(n-1));
     finSi;
Fin ;
```

**Exécution d'une fonction récursive :** L'appel de la fonction pour calculer la factorielle de la valeur 6 par exemple provoquera les appels suivant :

Fact(6) = 6 * Fact(5) → 720
= 5 * Fact(4) → 120
= 4 * Fact(3) → 24
= 3 * Fact(2) → 6
= 2 * Fact(1) → 2
= 1* Fact(0) = 1

**Support de cours**

# 5. Les sous-programmes en langage C

Le langage C implémente seulement la notion de fonction. L'implémentation des procédures n'existe pas en langage C mais il existe la possibilité de définir une procédure en utilisant la syntaxe de la fonction.

## a) Définition d'une fonction

La définition d'une fonction qu'on appelle corps de la fonction est de la forme suivante :

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

**Type-fonction :** Est le type de retour (résultat) de la fonction.

**Nom-fonction** : Est un identificateur permettant de nommer la fonction.

**Type-1…Type-n**: sont les types des paramètres de la fonction arg-1…arg-n

Une fonction renvoie sont résultat avec l'instruction

```
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.

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 puissance qui calcule $a^n$

```
int puissance (int a, int n){
     int p = 1;
     if (n == 0)
     return(1);
     for(i = 1,i<=n,i++)
     p = p*a ;
     return(p);
}
```

## b) Appel d'une fonction

L'appel d'une fonction se fait par l'expression. En execution l'appel de la fonction est remplacé par la valeur retournée.

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

**Exemple :** un programme qui calcule $x^n$ en appelant la fonction puissance.

```
#include<stdio.h>
int puissance (int a, int n){
     int p = 1;
     if (n == 0)
     return(1);
     for(i = 1,i<=n,i++)
     p = p*a;
     return(p);
}
```

```
int main(){
int n,x;
scanf("%d %d",&x,&n);
printf("x puissance n = %d ",
puissance(x,n)) ;
return 0;
}
```

## c) Les procédures en langage C

Une procédure est une fonction qui ne retourne pas de résultat. Pour implémenter une procédure en langage C il suffit de définir une fonction de type **`void`**. Dans une fonction de type `void` le mot clés `return` est facultatif, on peut l'ignorer comme on peut écrire `return;`

**Exemple :** une procédure qui affiche la valeur d'une variable entière passée en paramètre.

```
void afficher (int n){
     printf("n = %d \n",n);
     return;
}
```

## d) Paramètres passage par adresse :

En langage C, le mode de passage des paramètres par défaut est par valeur. Pour passer un paramètre d'une fonction en mode passage par adresse, il faut utiliser le type pointeur qui représente l'adresse mémoire d'une variable.

```
int n, *p ;
```

Dans la déclaration ci-dessus, n est une variable entière et p est une variable de type adresse d'une case mémoire de type entier.

Pour modifier la valeur de la case mémoire correspondant à l'adresse p on écrit :

```
*p = valeur ;
```

**Exemple** : une fonction permuter permettant de permuter les valeur de variable réelles.

```
void permuter (float *x, float *y){
float z;
     z = *x; //*x est le contenu de la case mémoire ayant x comme adresse.
     *x = *y;
     *y = z;}
```