Chapitre I : Procédures et Fonctions

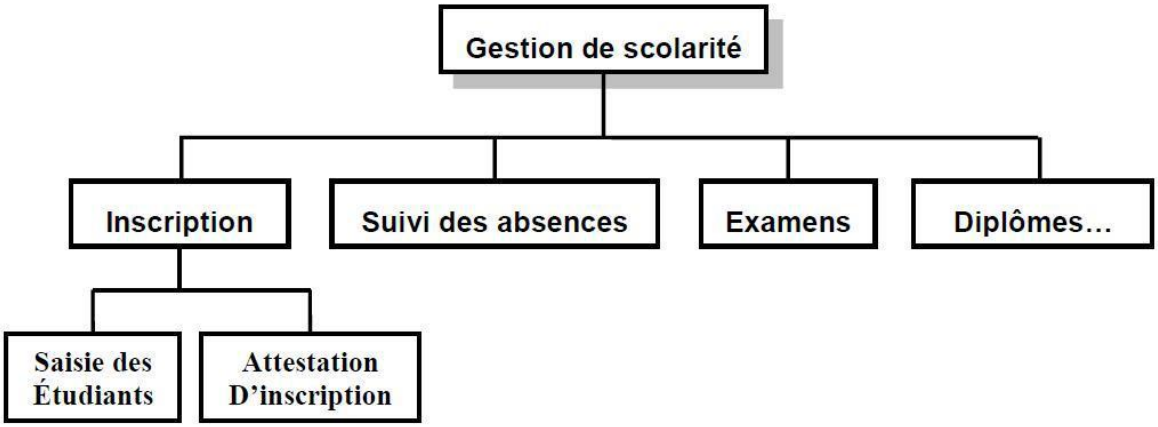
1. Introduction

Dans certains cas, on se retrouve devant une série d'actions qui revient plusieurs fois. Dans ce cas il serait préférable de l'écrire une seule fois, et de l'utiliser autant de fois.
En programmation, ce phénomène s'appelle ***la réutilisabilité***. Cette dernière est « *l'aptitude d'un algorithme à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu* ». (Jacques TISSEAU, 2009)

Cet usage est pratique lorsqu'on fait face à un algorithme complexe. Dans ce cas, Le programmeur décompose donc le problème en mini-blocs ayant chacun un rôle bien précis. Ces mini blocs sont des sous-programmes (sous-algorithmes) ou de morceaux appelés ***Procédures*** ou ***Fonctions***.

La structuration d'un programme par morceaux (modules) est la base de la programmation structurée et modulaire.

A titre d'exemple, un programme de gestion de scolarité peut être découpé en plusieurs modules : inscription, suivi des absences, examens, diplômes, etc. (F. Mguis, 2016)



2. Avantages d'utilisation des procédures et des fonctions

- Les programmes deviennent plus lisibles est plus organisés.
- Optimisation du code : le nombre d'instructions se diminue.
- Le contrôle et la maintenance des programmes deviennent plus aisés (vérification, correction et mise à jour)

3. Les procédures

3.1.Définition

Une procédure est une suite d'instructions décrivant une action simple ou composée, à laquelle on donne un nom, qui devient lui-même en quelque sorte un sous-programme.

3.2.Structure

Une procédure admet presque la même structure qu'un algorithme. Sa déclaration prend la forme suivante :

Procédure **Nom_de_la_procédure (Liste d'arguments : type)**
Les déclarations
DÉBUT
Instructions de la procédure
FIN

La première ligne s'appelle l'en-tête de la procédure. La liste d'arguments est une suite de données à échanger avec d'autres programmes.
La liste des paramètres (arguments) est facultative, on peut avoir des procédures sans arguments.
L'appel d'une procédure se fait par l'écriture de son nom dans le programme principal en lui passant les paramètres nécessaires

Exemple 1 : Procédure simple (sans paramètres) :

```
ALGORITHME exemple1
Variables                                   ← Variables Globales
X : string
Procédure TRAIT ( )                         ← Nom de la procédure (sans paramètres)
Variables
    i : entier                              ← Variable Locale
DÉBUT
    Pour i Allant de 1 A 20 Faire           ← Corps de la procédure
      Ecrire ("* ")
    FinPour
FIN

DÉBUT                                       ← Début de l'algorithme principal
  Lire (X)
  Ecrire (X)
  TRAIT( )                                  ← Appel de la procédure
FIN
```

Exemple 2 : Procédure avec paramètres :

```
Algorithme dessin
procédure ligne (nb : entier car : caractère)   ← Paramètres de la procédure
  Variables                                     ← Variable locale
    i : entier
DEBUT
    Pour i Allant de 1 A nb faire
    Ecrire (car)
    Fin pour
  Ecrire (saut de ligne) /* passage à la ligne */
FIN
DEBUT
  ligne(1, '*')
  ligne(4, '+')
  ligne(7, 'o')
FIN
```

4. Les Fonctions

4.1. Définition

Une fonction est une suite ordonnée d'instructions qui retourne une valeur (bloc d'instructions nommé et paramétré). C'est une procédure particulière qui admet un seul paramètre variable de type simple (entier, caractère, réel, etc.). Ainsi une fonction est tout simplement une procédure qui ne retourne qu'un seul résultat, une seule valeur, au programme appelant.
Il est obligatoire de préciser, dès le début le type de la fonction qui est en même temps le type du résultat à retourner.

4.2. Structure

Une procédure admet presque la même structure qu'un algorithme. Sa déclaration prend la forme suivante :

Fonction Nom_de_la_fonction (Paramètres formels) : type
Les déclarations
DÉBUT
<Séquence d'instructions>
Nom_de_la_fonction Résultat
Fin

L'appel d'une fonction se fait par l'écriture de son nom dans le programme principal en lui passant les paramètres nécessaires.

Exemple

```
Algorithme calcul
Var iables
   X :entier                                  ← Variable Globale
Fonction carre (a : entier) : entier          ← Nom de la fonction avec ses
Variables                                       paramètres et son type de retour
   result : entier                            ← Variable locale
DÉBUT
  result ← a*a                                ← Corps de la fonction
FIN

DEBUT
 LIRE (x)
          Appel de la fonction ECRIRE (carre (x))

FIN
```

Remarques

- Le nom de la fonction joue un double rôle, c'est à la fois l'identifiant de la fonction et une variable locale
- La fonction est un cas particulier de la procédure. La différence entre fonction et procédure se trouve à deux niveaux :
  - Au niveau du résultat, la fonction délivre un résultat et un seul.
  - Au niveau de l'appel, l'appel apparaît toujours dans une expression ou affectation.
- Une procédure peut ne pas posséder aucuns paramètres. Dans ce cas, elle réalise toujours la même action lorsqu'on l'invoque.

5. Pratique des procédures et fonctions en langage C

5.1. Les fonctions

Définition

type_de_retour nom_fonction **(liste-params)**
{
liste-declarations (optionnelle)
liste_instructions
}

Rem :

La liste d'instructions comprend **au moins** une instruction return (du type type_de_retour).

Appel

nom_fonction (liste-expressions)

Exemple :

Partie fonction

```
#include <stdio.h>
int puiss (int a, int n)   ← on déclare une fonction qui s'appel « puiss » qui
                             calcule la puissance entre deux entier a et b et qui
                             retourne un entier
  {
    int  i, result;        } ← variables locales
    result=1;

    for (i=1; i<=n;i++)    ← corps de la fonction
    result = result * a;
    return result;         ← la liste des instructions doit emboiter au moins une
                             instruction return (du type type_de_retour).
  }
```

Partie programme principal

```
 int main ()
 {
 int x,y;   ← variables globales

 printf("donner a:");
 scanf("%d",&x);
 printf("donner b:");
 scanf("%d",&y);
printf ("le resultat de %d puiss %d = %d \n",x,y,puiss (x,y));
return 0;                  ← Appel de la fonction
}
```

5.2. Les procédures

Définition

En langage C, les procédures sont vues comme « des fonctions qui ne renvoient rien ». Une procédure est une fonction renvoyant ***void***, dans ce cas return est appelé sans paramètre. De ce fait, ne comprenant qu'une fonction renvoie une valeur alors qu'une procédure ne renvoie pas de valeur, mais provoque un 'effet de bord' (écriture sur écran, provocation de son, etc.).

void nom_procédure **(liste-params)** (optionnelle)
{
liste-declarations (optionnelle)
liste_instructions
}

Appel

nom_procédure (liste-expressions)

Exemples

Procédure sans paramètres

```
#include <stdio.h>
void imprimer ()
{
printf("**********\n") ;
printf(" *****  \n");
}
int main()
  {
imprimer() ;
 imprimer() ;
  return 0 ;
}
```

Procédure avec paramètres

```
#include <stdio.h>
void imprimer (char x)
{
printf("%c%c%c%c%c%c%c%c\n",x,x,x,x,x,x,x,x) ;
printf(" %c%c%c%c  \n",x,x,x,x);
}
int main()
{
char x;
printf("donner x:");
scanf("%c",&x) ;
imprimer(x) ;
imprimer(x) ;
return 0 ;
}
```

6. Modes de passage de paramètres

Avant de décrire les modes de passage de paramètres nous devons distinguer les notions suivantes :

Variable globale

Est une variable définie dans l'en-tête du programme principal. Elle est utilisable dans n'importe quel sous-programme sans nécessité de redéfinition.
Toutefois, si dans un sous-bloc il existe une variable qui porte le même nom que la variable globale, alors c'est cette variable locale qui sera considérée à l'intérieur du sous-bloc.

Variable locale

Est une variable définie à l'intérieur d'un sous-programme (procédure ou fonction). Sa portée (visibilité) est limitée au bloc qui la contient. Il serait donc erroné de l'utiliser dans le bloc principal ou dans un autre sous-bloc appartenant à l'algorithme.

Paramètres fictifs (formels)
Ce sont les paramètres qui figurent dans l'entête de la déclaration de la procédure. Ils sont utilisés dans les instructions de la procédure et la seulement. Ils correspondent à des variables locales.

Paramètres effectifs (réels) :
Ce sont les paramètres qui figurent dans l'instruction d'appel de la procédure. Ils sont substitués aux paramètres formels au moment de l'appel de la procédure.

Algorithme Echange
Variables
a,b,I :Entier
Procédure Echange (x : entier, y : entier)
Variables
tampon :entier
Début
tampon ← x
x← y
y← tampon
Fin Procédure
Début
Lire (a)
Lire(b)
I←a-b
Ecrire (I)
Echange (a ,b)
I←a-b
Ecrire (I)
Ecrire(a)
Ecrire(b)
Fin
	Paramètres fictifs (formels) : x et y
	
	Paramètres effectifs (réels) : a et b
	
	Variables globales : a,b et I
	Variables locales : tampon
		#include <stdio.h>
		void Echange (int **x**, int **y**)
		{
		int tampon;
		tampon =*x ;
		*x=*y ;
		*y=tampon ;
		}
		
		int main ()
		{
		int a,b,I;
		printf("donner a:");
		scanf("%d",&a);
		printf("donner b:");
		scanf("%d",&b);
		I=a-b;printf("%d \n",I);
		Echange(a, b);
		I=a-b;printf("%d \n",I);
		printf("a= %d \n",a);
		printf("b= %d \n",b);
		return 0 ;
		}

Procédure Echange (x :entier, b :entier)

Echange (a ,b)

void Echange (int x, int y)

Echange(a,b);

Le paramètre formel « **x** »représente le paramètre effectif « **a** » et le paramètre formel « **y »** représente le paramètre effectif « **b** ».

6.1. Fonctionnement et utilisation des paramètres

Lorsque la déclaration d'un sous-programme comporte des paramètres formels, ceux-ci doivent être représentés chacun par son identificateur ainsi que par son type.

Ainsi pendant la construction de l'algorithme principal, il faudra toujours veiller à ce que chaque appel du sous-programme soit suivi d'une liste de paramètres effectifs correspondant (en nombre, rang, et type) à la liste des paramètres formels.

6.2. Mécanisme de passage de paramètres

Le rôle principal des paramètres est de transmettre les données entre le programme et la fonction ou la procédure appelée. En fait, il existe deux modes de passages : le passage de paramètre par valeur et le passage de paramètre par adresse.

6.2.1. Passage de paramètres par valeur

Le code appelé dispose d'une copie de la valeur qu'il peut modifier sans affecter son contenu initiale. C'est le mode de passage par défaut. Dans ce cas, la procédure ou la fonction lors de son appel travail uniquement sur une copie des variables effectives ou réelles. De ce fait, le contenu des paramètres effectifs ne peut pas être modifié par les instructions de la fonction ou de la procédure car nous ne travaillons pas directement avec, mais avec une copie.

Exemple :
Dans le programme ci-dessus qui contient la procédure **Echange** si on donne respectivement aux variables **a** et **b** les valeurs **8** et **3**. Le programme affiche **I=5** (le résultat de a-b).
Après l'appel de la procédure Echange, normalement **a** devient **3** et **b** devient **8** après permutation et **I** devient -**5**. Mais comme le passage utilisé dans cet exemple est un passage par valeur **a** et **b** reste sans changement et ils gardent leurs valeurs et par conséquent le résultat I=a-b reste tel qu'il est aussi.

6.2.2. Passage de paramètres par adresse (variable)

Le code appelé dispose d'une information lui permettant d'accéder en mémoire à la valeur que le code appelant cherche à lui transmettre. Il peut alors modifier cette valeur là où elle se trouve ; le code appelant aura accès aux modifications faites sur la valeur. Dans ce cas, le paramètre peut aussi être utilisé comme un paramètre de sortie.

Exemple :
Pour avoir une permutation réelle de valeur entre a et b de l'exemple précédent, on doit passer leurs adresses plutôt que leurs valeurs. De ce fait et pendant l'exécution des instructions de la procédure, le paramètre formel fait référence au même contenant variable que celui désigné par le paramètre effectif.

```
Algorithme Echange
   Variables
        a,b,I :Entier
    Procédure Echange (VAR x : entier, VAR y : entier)
       Variables
         tampon :entier
    Début
     tampon ← x
     x← y
     y← tampon
    Fin Procédure
  Début
  Lire (a)
  Lire(b)
  I←a-b
  Ecrire (I)
  Echange (a ,b)
  I←a-b
  Ecrire (I)
  Ecrire(a)
  Ecrire(b)
Fin
```

```
#include <stdio.h>
void Echange (int *x, int *y)
{
int tampon;
tampon =*x ;
*x=*y ;
*y=tampon ;
}

int main ()
{
int a,b,I;
printf("donner a:");
scanf("%d",&a);
printf("donner b:");
scanf("%d",&b);
I=a-b;printf("%d \n",I);
Echange(&a,&b);
I=a-b;printf("%d \n",I);
printf("a= %d \n",a);
printf("b= %d \n",b);
return 0 ;
}
```

Si on veut changer réellement la valeur d'une variable lors de l'appel d'une fonction ou une méthode on ajoute **&** devant son nom dans l'appel et ***** devant son paramètres formel équivalent

Si on exécute ce programme et on donne respectivement **8** et **3** à **a** et **b** le programme affiche les résultats suivants :

I=5
I= -5
a=3
b=8

Ce qui explique le ***changement réel*** des valeurs de a et b

7. La récursivité

En programmation, nombreux sont les problèmes qu'on résout en répétant plusieurs fois des séquences d'instructions.

Certains langages sont munis de structures de contrôles répétitivesfs. C'est le cas notamment pour la langage C, qui dispose des boucles `pour` (`for`) et `tant que` (`while`).

Mais certains problèmes se résolvent simplement en résolvant un sous problème de même nature, mais plus simple…Cette méthode de résolution s'appelle la *récursivité*.

7.1.Définition

Une fonction récursive est une fonction qui s'appelle elle-même. Chaque appel à la fonction est indépendant des autres, avec ses propres variables.

L'exemple le plus classique d'emploi de la récursivité est l'écriture de la fonction factorielle. Pour rappel, la factorielle d'un nombre *n* est définie comme *n* fois la factorielle du nombre *n*-1, et la factorielle de 1 est 1.

```
Alrotithme Fact
Variables
  n : entier
Fonction factorielle (n : entier) :entier
  Variable f :entier
  Début
      Si n == 1 Alors
          f = 1
      Sinon
          f = factorielle (n-1) * n
      Fin Si
    return f
Fin Fonction
Début
    n ← 13
    Ecrire (n,' ! = ', factorielle (n))
Fin
```

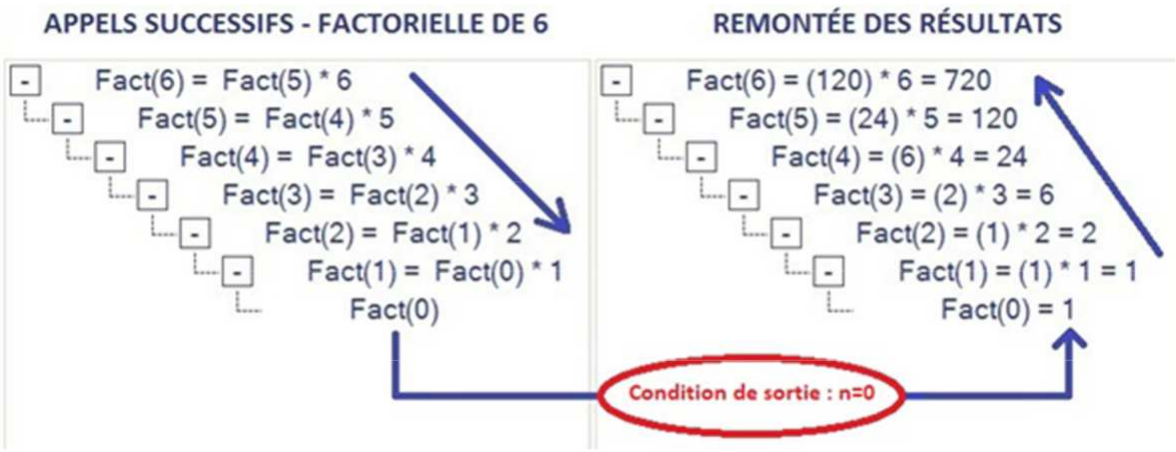
```
#include <stdio.h>

/* Fonction qui calcule n! */
int factorielle(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorielle(n - 1);
}

int main(void)
{
  int n=13;
  printf("%d! = %d\n", n, factorielle(n));
  return 0;
}
```

7.2. Pile d'exécution

2. Lorsqu'on fait des appels de fonctions, la pile d'appels de fonctions permet de mémoriser l'endroit du programme où la fonction a été appelée, avec les paramètres et la valeur de retour. Lorsqu'une fonction a terminé son exécution, elle est dépilée.

3. Lors de l'exécution d'une fonction récursive, on appelle la même fonction plusieurs fois, ce qui produit plusieurs empilages :

Exemple



N.B :

3. L'écriture itérative de la fonction factorielle est très simple et dans ce cas, on préfère l'utiliser plutôt que la version récursive.
4. Un usage irréfléchi de fonctions récursives peut poser des problèmes d'exécution sur certaines implémentations. En effet, si la récursivité est profonde i.e. les appels récursifs sont nombreux, les fonctions sont appelées et ne retournent pas immédiatement leur résultat donc les appels s'accumulent. Ces appels sont souvent stockés dans une zone mémoire appelée "pile" et qui est de taille limitée. Si cette taille est dépassée, on obtient une erreur à l'exécution ("une explosion de la pile"). Par exemple, si on exécute la fonction ci-dessus avec une valeur de n très grande, on peut obtenir une erreur à l'exécution : ainsi, le code ci-dessous

```
#include <stdio.h>
/* Retourne le produit n*x où n>=0 */
int produit(int n, int x) {
  if (n > 0)
    return produit(n - 1, x) + x;
  else
    return 0;
}
int main(void)
{
  int n = 500000, x = 5;

  printf("%d * %d = %d\n", n, x, produit(n, x));
  return 0;
}
```

S'exécute de la manière suivante :

```
4.    gcc -W -Wall -std=c99 -pedantic -o x recursifPile.c
5.    ./x
Erreur de segmentation
6.
```

En fait, le code récursif utilisé peut-être dérécursifié de la manière suivante :

```
/* derecursifier.c */
#include <stdio.h>

/* Retourne le produit n*x où n>=0
*/ int produit_bis(int n, int x) {

  int i = 0;
  int temp = 0;

  for (i = 0; i < n; i++)
    temp = temp + x;

  return temp;
}

int main(void)
{
  /* on suppose que la valeur 500000 tient dans un
  i int n = 500000, x = 5;

  printf("%d * %d = %d\n", n, x, produit_bis(n,
  x)); return 0;
}
```

qui affiche le message suivant : `500000 * 5 = 2500000`

7.3. Règles de conception d'un algorithme récursif

Règle 1 :

Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne doit pas contenir d'appels récursifs, sinon il y a risque de cercle vicieux et de calcul infini. Les cas non récursifs d'un algorithme récursifs sont appelés *cas de bases*. Les conditions que doivent satisfaire les données dans ces cas de bases sont appelées *conditions de terminaison.*

Règle 2 :

Tout appel récursif doit se faire avec des données plus proches de données satisfaisant les conditions de terminaison.

7.4.Type de récursivité

7.4.1. Récursivité simple ou linéaire

Un algorithme récursif est *simple* ou *linéaire* si chaque cas qu'il distingue se résout en au plus un appel récursif. Ainsi l'algorithme de calcul de n! est récursif simple.

7.4.2. Récursivité multiple

Un algorithme récursif est *multiple* si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs.

Exemple :

Fonction combinaison (N : entier, P : entier) : entier
Début
Si (P=0 ou N=0) **Alors**
Return 1
Sinon
Return (combinaison (N-1, P) + combinaison (N-1, P-1))
FinSi
Fin Fonction

7.4.3. Récursivité croisée ou mutuelle

Deux algorithmes sont *mutuellement récursifs* si l'un fait appel à l'autre et l'autre à l'un.

Parité d'un entier P(n) = prédicat de test de parité de l'entier n. I(n) = prédicat de test d'imparité de l'entier n. Solution mutuellement récursive

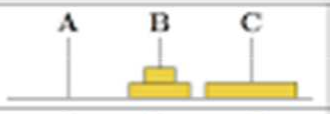
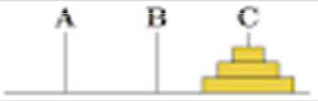
Fonction P (n : entier) : Booléen
Début
Si n= 0 **alors**
P(n) = vrai
Sinon
P(n) = I(n−1)
fin si
Fin Fonction

Fonction I(n :entier):Booléen
Début
Si n= 0 **alors**
I(n) = faux
Sinon
I(n) = P(n−1)
fin si
Fin Fonction

Exemple : tours de Hanoi

C'est un jeu de réflexion qui consiste à déplacer un nombre donné de disque d'une tour "départ" à une tour "arrivée", à l'aide d'une troisième tour "intermédiaire" et ceci en respectant deux règles :

5. Ne déplacer qu'un seul disque à la fois.
6. Ne pas placer un disque sur un autre plus petit.

Mouvement	Position	Mouvement	Position
Position initiale		4 : A vers C	
1 : A vers C		5 : B vers A	
2 : A vers B		6 : B vers C	
3 : C vers B		7 : A vers C	

Solution

```
#include <stdio.h>
void hanoi (int n, char A, char B, char C)
{
   if(n==1)
   printf("deplacer %c vers %c\n",A,C);
   else
   {
     hanoi(n-1,A,C,B);
     printf("deplacer %c vers %c\n",A,C);
     hanoi(n-1,B,A,C);
   }
}

main()
{
  char a='a',b='b',c='c';
  int n;
   while(1)
  {
    printf("Entrer n : ");
    scanf("%d",&n);
    printf("\n");
    hanoi(n,a,b,c);
  }
}
```