L1-Informatique – Section 3 – Algorithmique et Structures de Données 2 (ASD2)
Chapitre 1 : Les Sous-Programmes (Fonctions et Procédures)- Langage Algorithmique et Langage C

Contenu

1. *Introduction*
2. *Définitions*
3. *Les variables locales et les variables globales*
4. *Le passage des paramètres*
5. *La récursivité*

1. Introduction :

a. *Notions de Programme principale et Sous-programmes*

En matière de programmation, nous réalisons des algorithmes et programmes qui résolvent des problèmes divers. Lorsque le problème à résoudre est relativement simple et donc de solution simple aussi, la solution (algorithme ou programme) est aussi simple, voire, évidente. Dans ce cas, la solution (algorithme ou programme) est réalisée en un seule 'Bloc d'instructions'. C'est ce genre de solution (algorithme ou programme) qui ont été réalisées jusqu'à présent dans ce cours (ASD1).

Dans ce Chapitre1 de la matière ASD2, nous introduisons des solutions (algorithmes ou programme) qui comprennent :

- Un programme/algorithme Principal

  Ce programme contiendra une ossature générale de la solution globale S du problème posé P et utilise des programmes/algorithmes partiels pour résoudre des sous-problèmes de P. Nous appelons cet algorithme/programme Principal : SP (Solution Principale).
- Un ou plusieurs programmes/algorithmes partiels, dits : Sous-programmes / Sous-algorithmes où, chaque sous-programme résout une partie du problème P. Nous appelons ces sous programmes : SP1, SP2, etc. La Figure 5.1 illustre les deux situations schématiquement.

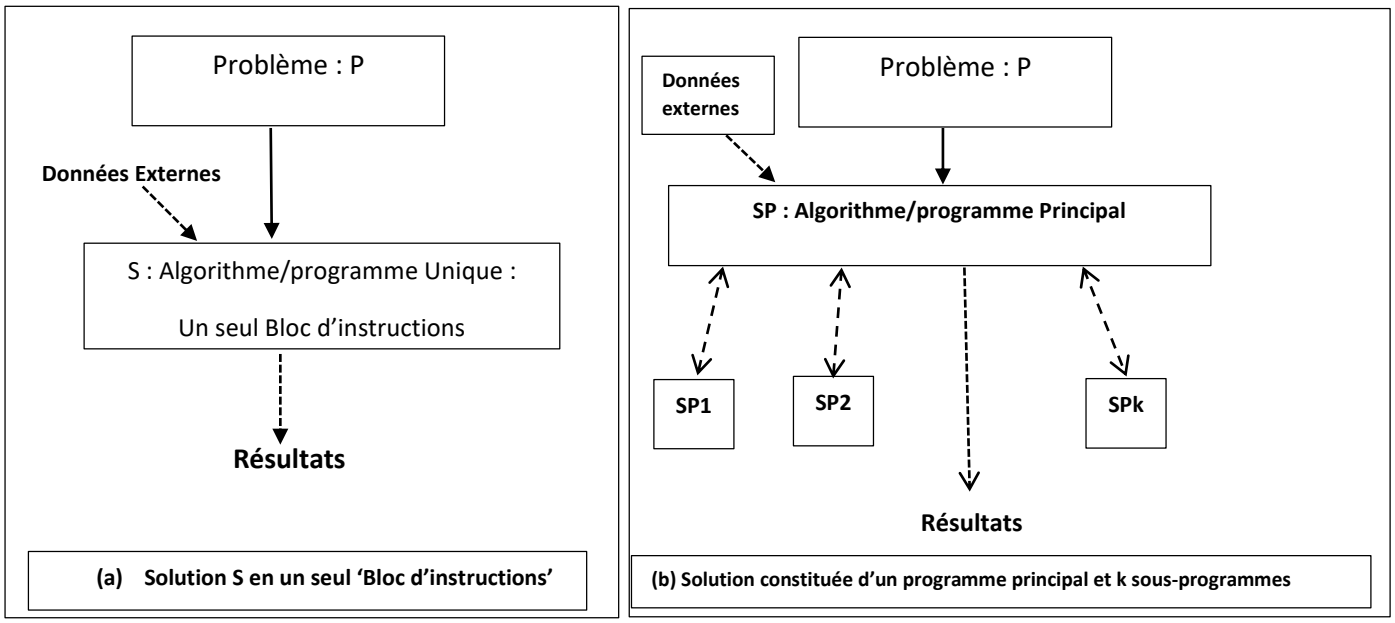

Figure 5.1 Illustration schématique de programmes en un 'Bloc d'actions' unique (a) et (b) de programmes constitués d'un Programme principale et de 1 ou plusieurs sous-programmes

D'autre part, les sous-programmes/algorithmes sont de deux catégories :

- Les Fonctions
- Les Procédures

Ces deux concepts seront présentés dans ce chapitre.

b. Motivations pour les sous-programmes :

Il est clair que la solution avec sous-programmes fait exactement le même travail pour résoudre le problème posé, mais :

- de façon plus compacte.
- De façon plus claire à comprendre, ce qui facilite la réalisation et la maintenance des programmes.
- Remarquez aussi que, il n'est pas obligatoire qu'un sous-programme réalisé soit utilisé plusieurs fois, puisque les sous-programmes sont utilisés surtout pour réduire la complexité (difficulté) des problèmes posés.
- Remarquez aussi, que depuis la matière ASD1, nous utilisions déjà des sous programmes. Exemples : *scanf*() et *printf*() sont des sous-programmes prédéfinies dans le langage C et sont logées dans la bibliothèque que nous utilisons depuis l'ASD1 : <*stdio.h*>.

Nous détaillons ces notions dans les points suivants de ce cours.

2. Définitions des Sous-Programmes : Fonctions et Procédure

a. **Définitions** :

i. *Programme/algorithme Principal* :

Etant donné un problème P, la solution S de P est constituée d'un programme principal PP et, éventuellement, d'un ou plusieurs sous-programmes SP1, SP2, etc…, qui résolvent des parties de P.

- Au lancement, c'est le programme principal qui s'exécute en premier
- Le programme Principal contient la structure générale de la solution.
- Il invoque (appelle, call) ses Sous-Programme selon la logique de la solution S.

ii. *Sous-Programmes* :

Un sous-Programme SP d'un programme principal PP est un programme qui résout une partie du problème posé P. Comme mentionnées, les sous-programmes sont de deux types :

- Les *Fonctions*
- Les *Procédures*

Nous effectuons alors une *comparaison* entre ces deux outils algorithmiques et de programmation.

i. Points communs :

- Les Fonctions, comme les Procédures, sont des *sous-programmes* qui résolvent une partie d'un problème (sous-problème) du problème principal P.
- Les Fonctions, comme les Procédures, reçoivent des *paramètres* (entrées) et renvoient des résultats (sorties). Ces paramètres sont dits : *Paramètres formels* de la Fonction ou de la Procédure.
- Les Fonctions, comme les Procédures, sont constituées, comme les programmes/algorithmes principaux, de trois parties :

  Entête : Pour déclarer la Fonction ou la Procédure

  Zone des variables : Pour déclarer les variables de la fonction ou la procédure

  Zone des actions : Qui contient les actions de la Fonction ou de la Procédure qui résolvent le sous-problème en question.

- Les Fonctions, comme les Procédures, sont *appelées* (*invoquées, Called*) dans d'autres sous-programmes ou dans le programme principal en utilisant des *paramètres d'appel*. Ces paramètres dont dits : *Paramètres effectifs*.

ii. Points de différence :

- Les Fonctions se distinguent par les points suivants :

- La Fonction retourne toujours 1 et 1 seul résultat de type simple : entier, réel, caractère, …
- La Fonction possède toujours *au moins 1 paramètre d'appel*.

- Les Procédures se distinguent par les points suivants :

- La Procédure peut avoir *0, 1 ou plusieurs paramètres*.
- La Procédure n'est pas typée.

b. *Illustrations* :

Nous illustrons, d'abord, le principe des sous-programmes par des exemples simples.

i. *Exemple1* :

Soit n un nombre entier. Nous voulons calculer :

- La somme des nombres impairs dans l'intervalle 1…n et
- La somme des nombres pairs dans l'intervalle 1…n

- *Solution en un seul bloc : Langage Algorithmique :*

```
Algorithme SomPairImpair
Données n : entier
Résultats simpair, spair : entier
Variables i : entier
Début
        Ecrire('Donnez n=') ;
        Lire(n)
        simpair=0
        pour(i=1 jusqu'à n, pas=2) faire
             simpair = simpair+i
        finpour
        écrire('' Somme des nombres impairs='',simpair)

        spair=0
        pour(i=2 jusqu'à n, pas=2) faire
             spair = spair+i
        finpour
        écrire('' Somme des nombres pairs='',spair)
  Fin
```

- *Solution en un seul bloc : Langage C :*

```
#include<stdio.h>
int main(){
  int n;
  printf(" Donnez n = ");
  scanf("%d",&n);
  int i,spair=0, simpair=0;

  for(i=1;i<=n;i+=2){
             simpair +=i;
             printf("\n Somme des nombres impairs de 1 A %d = %d",i,simpair);
             }
  for(i=2;i<=n;i+=2){
             spair +=i;
             printf("\n Somme des nombres pairs de 1 A %d = %d",i,spair);
             }
  printf("\n Somme des nombres impairs de 1 A %d = %d",n,simpair);
  printf("\n Somme des nombres pairs de 1 A %d = %d",n,spair);
}
```


Exécution :

```
Donnez n = 10

Somme des nombres impairs de 1 A 1 = 1
Somme des nombres impairs de 1 A 3 = 4
Somme des nombres impairs de 1 A 5 = 9
Somme des nombres impairs de 1 A 7 = 16
Somme des nombres impairs de 1 A 9 = 25
Somme des nombres pairs de 1 A 2 = 2
Somme des nombres pairs de 1 A 4 = 6
Somme des nombres pairs de 1 A 6 = 12
Somme des nombres pairs de 1 A 8 = 20
Somme des nombres pairs de 1 A 10 = 30
Somme des nombres impairs de 1 A 10 = 25
Somme des nombres pairs de 1 A 10 = 30
--------------------------------
```

Constatations : Comme il peut être vu dans cet algorithme, il y a deux 'Blocs d'instructions' qui répètent pratiquement les mêmes actions :

'Bloc d'instructions 1' : Ce Bloc, calcule la somme des nombres impairs de 1 à n.

'Bloc d'instructions 2' : Ce Bloc, calcule la somme des nombres pairs de 2 à n.

Grâce à la notion de *sous-programme*, nous pouvons réaliser une meilleure solution, comme suit :

ii. Définition : *Déclaration de Fonctions en langage algorithmique* :

En langage algorithmique, une fonction est déclarée comme suit :

```
Fonction Nom-Fonction(Liste-de-Paramètres) : Type-de-Fonction
Variables : Listes-de-variables
 Début
    Action1
     ...
    Action N
 Nom-Fonction ← Expression
 Fin
```

Où :

- *Fonction Nom-Fonction(Liste-de-Paramètres) : Type-de-Fonction* est l'entête de la fonction
  Son rôle est de définir la fonction (Données – Résultats)
- *Liste-de-Paramètres : ensemble des paramètres de la fonction : Param1 :Type1, ...*
- *Ces paramètres sont dits paramètres formels.*

- Par la suite, au moment de l'utilisation de la fonction, les *paramètres passés à la fonction* (dits, *paramètres effectifs*) doivent vérifier les conditions suivantes :
- Leurs nombres doit être égal au nombre de paramètres formels de la fonction
- Les types des paramètres effectifs passés à la fonction doivent être les mêmes que ceux des paramètres formels, dans l'ordre de définition de la fonction.

Variables : Listes-de-variables : est la zone où les variables nécessaires à la fonction sont déclarées.

```
Début
        Action1
         ...
        Action N
    Nom-Fonction ← Expression
    Fin
```

Cette partie est la zone des actions de la fonction.

En particulier, l'action : *Nom-Fonction ← Expression* est obligatoire pour toute fonction. Son rôle est de *retourner* une valeur au point d'appel de la fonction.

iii. Déclaration de fonctions en langage C :

En langage C, une fonction est déclarée comme suit :

```
Function-Type Function-Name(Liste-de-paramètres){ :  // Entête
Variables declaration          // Variables
Instruction1 ;                 // Instructions
  ...
InstructionN ;
Return Expression ;            // return value (result)
}
```

Exemple : Calcul des sommes des nombres impairs (1 à n) et des nombres impairs (2 à n):

- *Solution utilisant des sous-programmes en langage algorithmique (Fonction) :*

```
fonction sompas2(v1 : entier, v2 : entier) : entier      // Sous-programme de type fonction
Variables i, s: entier
 Début
    s=0
    pour(i=v1 jusqu'à v2, pas=2) faire
        s = s+i
    finpour
 sompas2 ← s
 fin

Algorithme somPairImpair2                    // Programme principale
Données n: entier
Résultats: spair, simpair: entier
Variables
 Début
        Ecrire('Donnez n=') ;
        Lire(n)
                    // appel de la fonction pour les nombres impairs
   Ecrire('' Somme des nombres impairs = '', sompas2(1,n))

                              // appel de la fonction pour les nombres pairs
   Ecrire('' Somme des nombres pairs = '', sompas2(2,n))
 fin
```


Constatations : Comme il peut être remarqué, la solution est beaucoup plus compacte et plus claire.

La fonction sompas2 possède alors :

- Un type : entier
- Deux paramètres formels : v1 : entier et v2 : entier
- Donc, à son utilisation, il faut donner deux paramètres effectifs de mêmes types que ceux des paramètres formels. Exemple : som ← sompas2(15, 95)
- La fonction retourne une valeur de type entier
- L'algorithme principal utilise la même fonction sompas2 avec des paramètres effectifs différents :

sompas2(1,n) pour la somme des nombres impairs

sompas2(2,n) pour la somme des nombres pairs.

- Programme équivalent de cet algorithme, en langage C :

```
#include<stdio.h>
int sompas2(int v1, int v2){
  int i,s;
  s=0;
  for(i=v1;i<=v2;i+=2){
            s +=i;
             }
  return s;
}

int main(){
  int n;
  printf(" Donnez n = ");
  scanf("%d",&n);
  printf("\n Somme des nombres impairs de 1 A %d = %d",n,sompas2(1,n));
  printf("\n Somme des nombres pairs de 1 A %d = %d",n,sompas2(2,n));
}
```

Exécution :

```
Donnez n = 10

Somme des nombres impairs de 1 A 10 = 25
Somme des nombres pairs de 1 A 10 = 30
--------------------------------
```

Dans cette solution, nous avons :

i. Un sous-programme de type *fonction* :

```
int sompas2(int v1, int v2){
          int i,s;
          s=0;
          for(i=v1;i<=v2;i+=2){
                    s +=i;
                     }
          return s;
}
```

Dans cette fonction :

- *int sompas2(int v1, int v2) : est l'entête de la fonction* (ou encore, la *définition de la fonction*).
- *sompas2* : est le *nom de la fonction*. Il sert à définir la fonction et aussi pour l'utiliser par la suite.
- Dans cet entête, *int sompas2* : indique que la fonction retourne un nombre *integer*.
- *int v1 et int v2* : ces déclarations indiquent que la fonction possède *deux paramètres v1 de type integer et v2 de type integer*. Donc, au moment de l'utilisation de la fonction, il faut lui fournir *deux valeurs (constantes, variables ou expressions) de type entier*.

- Zones des déclarations :

  int i,s; : La fonction a besoin de deux variables de type integer: i et s.

- *Zone des instructions* :

```
s=0;
for(i=v1;i<=v2;i+=2){
        s +=i;
        }
return s ;
```

Cette section représente le corps de la fonction (instructions de réalisation de la fonction).

En particulier, l'instruction : *return s ;* indique que la fonction retourne comme résultat un nombre entier (représenté dans la fonction par la variable *s)*.

Il faut signaler encore que la fonction retourne toujours *1 et 1 seul résultat*.

ii. Un programme principal :

```
int main(){
        int n;
        printf(" Donnez n = ");
        scanf("%d",&n);
        printf("\n Somme des nombres impairs de 1 A %d = %d",n,sompas2(1,n));
        printf("\n Somme des nombres pairs de 1 A %d = %d",n,sompas2(2,n));
}
```


Dans ce *programme principal*, il y a :

- Déclaration des variables : *int n ;*
- Lecture de n ;
- Ecriture des résultats (somme des nombres impair, puis celle des nombres pairs)
- Remarquez l'utilisation de la fonction sompas2 :

  Pour les nombres impairs : *sompas2(1,n)*

  Pour les nombres pairs : *sompas2(2,n)*

  Dans les deux cas, il y a deux paramètres (dits *effectifs*)

  Mais, pour le paramètre 1, les nombres impairs sont calculés à partir de 1.

  Pour les nombres pairs, ils sont calculés à partir de 2.

iii. *Définition* (*Procédure*) :

Comme les *Fonctions*, une *Procédure* est un sous-programme qui résout un sous-problème (de taille réduite) du problème principal. La procédure possède alors les mêmes éléments que les programmes (algorithmes) principaux :

- Une entête : *Procédure* NomProc(Liste-de-Paramètres)
- Une zone déclaration des données
- Une zone d'actions.

Où :

- *NomProc*: est le nom de la Procédure. Il est utilisé pour invoquer la procédure (appel, call)
- Liste-de-Paramètres : est un ensemble de paramètres passés à la procédure pour effectuer son travail. Le nombre de paramètres des procédures est variables : 0, 1 ou plusieurs.

  Nombre de paramètres = 0 → La procédure peut : effectuer des calculs et/ou afficher un ou plusieurs messages et/ou agir sur des variables qui lui sont accessibles (portée des variables).

  Nombre de paramètres = 1 ou plusieurs → La procédure utilise ces paramètre et, éventuellement, affiche des messages ou agit sur des variables qui lui sont accessibles (portée des variables)..
- Les sous-programmes de catégorie *Procédure* ne sont pas typés (ne possèdent pas de type, comme les fonctions qui le sont obligatoirement)

Exemple : Ecrire une *procédure* qui reçoit en paramètre un nombre entier positif n et affiche tous les nombres entiers ≤ n.

- Solution en langage algorithmique :

```
Procédure Entier1n(n : entier)      // entête
                                    // Nom : Entier1n,   Paramètres : n : entier

Variables i : entier                // déclaration des variables de la procédure
                                    // ici,  i : entier

Début                               // Zone Actions
      Pour (i=1 jusqu'à n) faire
          Ecrire(i)
      Finpour
Fin
```

Remarque :

- Dans cette procédure, l'action *Ecrire*() est aussi une procédure que nous utilisons régulièrement.
- Il n y a pas besoin de *Lire(n)*, puisque n est passé en paramètre à la procédure *Entier1n*
- Mais la valeur de n est initialisée par lecture ou affectation dans l'algorithme principal.
- L'utilisation de cette procédure (invocation, call) peut se faire alors comme suit :

```
Algorithme nbr1n
Données n : entier
Résultats   // tous les nombres de 1 à n
Variables
Début
    Lire(n)
    Entier1n(n)                    // Invocation (utilisation, appel, Call) de la
                                   // Procédure Entier1n, avec le paramètre n
Fin
```

- Programme équivalent en langage C :

```
#include<stdio.h>
void Entier1n(int n){              // Procédure ( void: ⇒ la procédure est non typée)
        int i;
        for(i=1;i<=n;i++){
          printf("%3d",i);
        }
}

int main(){                        // Programme Principal
 int n;
 printf("Donnez n = ");
 scanf("%d",&n);
 Entier1n(n);
}
```

Exécution :

```
Donnez n = 15
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
--------------------------------
```

3. *Modes de passation des paramètres à un sous-programme :*

Nous considérons les deux modes les plus connus et les plus utilisés de passation des paramètres aux sous-programmes (Fonctions et Procédures) :

- *Passation par valeur*
- *Passation par adresse*

i. Passation des paramètres par *valeur* :

C'est le mode couramment utilisé. Il consiste à mettre dans la position du *paramètre effectif* concerné une valeur (Constante, variable ou Expression) de même type que le type du paramètre formel concerné.

Dans ce mode, le paramètre *formel concerné* ne contient aucun autre symbole spécial (pour indiquer le *mode passation par valeur*).

Dans ce mode, le sous-programme reçoit *une copie de la valeur du paramètre effectif* (lorsqu'il s'agit d'une variable). Cela veut dire que, le sous-programme peut utiliser la valeur qui lui est passée, mais, il ne peut pas modifier la valeur du paramètre effectif lui-même (lorsqu'il s'agit d'une variable).

Lorsque le paramètre effectif est une constante ou expression (en général), il y évaluation de l'expression et le résultat de l'évaluation (1 valeur) est passé au sous-programme.

Bien sûre, le type de la valeur de l'évaluation doit être le même que celui du paramètre formel concerné dans le sous-programme.

Exemple :

Comme déjà illustré par l'exemple précédent pour le calcul des sommes des nombres impairs et somme des nombres pairs dans l'intervalle 1…n, n étant un nombre entier, le sous-programme de type fonction *sompas2* est défini sur la base deux variables *v1* de type entier et *v2* de type entier.

```
int sompas2(int v1, int v2) {   // paramètres formels de la fonction : int v1, int v2
int i,s;                        // Les deux paramètres v1 et v2 sont définis en mode
                                // passation par valeur.
  s=0;
  for(i=v1;i<=v2;i+=2){
            s +=i;
            }
  return s;
}
```

Exemple d'utilisation de la fonction sompas2 :

Dans le programme principal, nous avons deux *utilisations* de la fonction *sompas2* :

```
// programme principal utilisant la fonction sompas2 :

int main(){
      int n;
      printf(" Donnez n = ");
      scanf("%d",&n);
// utilisation 1 : somme des nombres impairs
      printf("\n Somme des nombres impairs de 1 A %d = %d",n,sompas2(1,n));
// Utilisation 2 : somme des nombres pairs
      printf("\n Somme des nombres pairs de 1 A %d = %d",n,sompas2(2,n));
}
```

Utilisation 1 : somme des nombres impairs

```
printf("\n Somme des nombres impairs de 1 A %d = %d",n,sompas2(1,n));
```

Ici, les *paramètres effectifs* sont : pour *v1 : 1* et pour *v2 : n*

// Utilisation 2 : somme des nombres pairs

```
printf("\n Somme des nombres pairs de 1 A %d = %d",n,sompas2(2,n));
```

Ici, les *paramètres effectifs* sont : pour *v1 : 2* et pour *v2 : n*

Le mode passation des deux paramètres v1 v2 est *par valeur*. En effet, aucun symbole n'est attaché aux paramètres dans leurs définitions dans la fonction.

D'autre part, il peut être constaté aussi que :

- Le nombre de *paramètres effectifs* est le même que celui des paramètres formels : 2.
- *Les types* respectifs des *paramètres effectifs* sont les mêmes que ceux des *paramètres formels* : *int* et *int*.
- Le *paramètre effectif v1* est passé selon une *constante* (dans les deux utilisations : 1, puis 2)
- Le *paramètre effectif v2* est passé selon une *variable* (dans les deux utilisations : n , puis n).

Exemple 2 :

Nous illustrons encore ces notions par un deuxième exemple :

Dans ce deuxième exemple, il est demandé de :

i. Ecrire une fonction *fact(n)* qui calcule la factorielle n! du nombre entier n, n ≥ 1.
Nous donnons : n!= 1*2*3*…n, n>1.

ii. Utiliser cette fonction pour calculer : $C_n^p = \frac{n!}{(n-p)!*p!}$:

(C_n^p : est le nombre de combinaisons de p objets parmi n).

- Solutions en langage algorithmique :
- *Fonction* : *Factorielle de n* : *fact(n)* // paramètre n : passé par valeur

```
fonction fact(n : entier) :entier
var i,f : entier
f ← 1 ;
début
        pour i=2 jusqu'à n faire
              f ← f*i
        finpour
fact ← f                                  // f=1*2*3…*n
fin
```

- Algorithme principal :

```
Algorithme CPN
Données n,p : entier
Résultats combp : entier
Début
      Lire(n,p)
      combnp ← fact(n)/(fact(n-p)*fact(p))        //3 utilisations de fact:
                                                  // variable, expression, variable
      Ecrire("CPN =",combnp)
Fin
```

- Solution en langage C :

```
#include<stdio.h>
int fact(int n){                    // paramètre formel de fact: n de type int
int i,f=1;                          // passation de n: Par valeur
for(i=2;i<=n;i++){
  f *=i;
}
return f;                           // résultat retourné
}
int main(){
  int n,p;
  int combnp;
  printf("Donnez n et p - avec 1<= p <= n : ");
  scanf("%d   %d",&n,&p);
  combnp = fact(n)/(fact(n-p)*fact(p));        // trois utilisations de la function fact
                                               // Paramètres effectifs : n, n-p et p de type int.
  printf(" C %d %d = %d ",n,p,combnp);
}
```

Exécution :

```
Donnez n et p - avec 1<= p <= n : 8 3
 C 8 3 = 56
--------------------------------
```

Pour comparaison, voici le programme C équivalent, n'utilisant pas de sous-programmes :


```
#include<stdio.h>
int main(){
  int n,p;
  int f=1, g=1, h=1;
  int combp,i;
  printf("Donnez n et p - avec 1<= p <= n : ");
  scanf("%d   %d",&n,&p);

  for(i=1;i<=n;i++){                    // n!
   f*=i;
}

for(i=1;i<=(n-p);i++){                  // (n-p)!
  g*=i;
}

for(i=1;i<=p;i++){                      // p!
  h*=i;
}
  combp = f/(g*h);                              // resultat :CPN
  printf(" C %d %d = %d ",n,p,combp);
}
```

Evidemment, les deux programmes sont équivalents, mais la version utilisant le sous-programme *fact(n)* est plus court et plus clair.

ii. Passation des paramètres *par adresse* :

Ce mode concerne le passage d'une variable comme paramètre. Il faut alors se rappeler que nous avons rapporté dans la matière ASD1 qu'une variable est caractérisée par les propriétés suivantes :

- *Un nom*
- *Une adresse*
- *Un Type*
- *Une valeur*

Voici des illustrations :

Considérons les déclarations en langage C suivantes :

int x=4;

float y=-2.5 ;

char z= 'h' ;

Nous aurons alors la représentation en mémoire suivante de ces variables :

N° (Adresse)	Nom	Type	Valeur
1	x	int	4
2	y	float	-2.5
3	z	char	104 (Code ASCII de 'h')

Maintenant, ces variables peuvent servir de paramètres effectifs pour une fonction ou procédure.
Nous aurons alors deux possibilités en passant x, y ou z comme paramètre effectif :

- Nous passons la *valeur* de chaque variable : C'est le mode de passation par valeur évoqué dans le point précédent. Nous rappelons que, dans ce mode, nous utilisons seulement une copie de la variable au niveau du sous-programme. D'où, il est impossible de modifier la valeur d'origine de la variable depuis le sous-programme.
  Rappel : Utilisation du mode de passation par valeur des variables : Nous avons mentionné dans le point précédent qu'il suffit d'écrire le nom de la variable sans aucun symbole additif et nous avons vu plusieurs exemples de ce mode.

- Nous passons l'adresse (ici, symbolisée par les numéros : 1 pour x, 2 pour y et 3 pour z).
  Dans ce mode, le sous-programme utilise l'adresse de la variable-paramètre pour accéder à sa valeur dans le point d'appel. Conséquences : Le sous-programme peut modifier la valeur des variables passées comme paramètres effectifs aux sous-programmes.

Utilisation du mode de passation par adresse des variables :

Dans ce mode, il faut indiquer au niveau du sous-programme et au niveau du point d'appel que la variable est passée par adresse. Nous adoptons dans ce cours la notation du langage C. Dans ce langage, une variable passée en paramètre en mode adresse se réalise comme suit :

- Au niveau du sous-programme (paramètre formel) :

  Le passage en mode adresse d'une variable x (de type *int*, par exemple) au niveau du paramètre formel est noté : *int* *x : accès à x par son adresse *x.

- Au niveau de l'utilisation du sous-programme (paramètre effectif) :

  Le passage d'une variable x en mode adresse au niveau du paramètre *effectif* (utilisation) est noté : &x. (Envoi de l'adresse de x au sous-programme).

Exemple :

Ecrire une fonction *Fpp* qui reçoit en *paramètre formel* une variable *x* de type int et lui ajoute 1. Nous voulons que l'ajout se fasse au niveau de la *variable-paramètre effectif*.

Voici un programme en langage C qui répond à la question (*passation de a par adresse*)

```
#include<stdio.h>
int Fpp(int *x){
        *x = *x +1;
        return *x;
}
int main(){
        int a;
        a=5;
        printf("Avant appel de Fpp : a= %d \n",a);
        printf("Fpp de a = %d \n",Fpp(&a));
        printf("Apres utilisation de Fpp a= %d",a);
}
```


Exécution :

```
Avant appel de Fpp : a= 5
Fpp de a = 6
Apres utilisation de Fpp a= 6
--------------------------------
```

Illustration du même programme en passant a par *valeur*.

```
#include<stdio.h>
int Fpp(int x){
  x = x +1;
  return x;
}
int main(){
  int a;
  a=5;
  printf("Avant appel de Fpp : a= %d \n",a);
  printf("Fpp de a = %d \n",Fpp(a));
  printf("Apres utilisation de Fpp a= %d",a);
}
```

Exécution :

```
Avant appel de Fpp : a= 5
Fpp de a = 6
Apres utilisation de Fpp a= 5
--------------------------------
```

Exemple 2 : Nous considérons un deuxième exemple, qui concerne cette fois-ci, l'utilisation d'une procédure en langage C qui permet d'effectuer la permutation des valeurs de deux nombres donnés a et b. Remarquez que cet exercice a été traité dans les séries de TDs de l'ASD1, avec une solution en un seul '*Bloc d'instructions*'.

Solution en langage C :

```
#include<stdio.h>
void permute(int *a, int *b){ // paramètres formels a et b : reçus par adresse
        int c;
        c=*a;          // a et b sont aussi utilisée via leurs adresses respectives
        *a=*b;
        *b=c;
}
int main(){
        int x,y;
        x=15;
        y=-49;
        printf("Avant Permute : x= %d  y = %d \n",x,y);
        permute(&x,&y);      // paramètres effectifs x et y envoyés par adresse
        printf("Apres Permute : x= %d  y = %d",x,y);
}
```

Exécution :

```
Avant Permute : x= 15  y = -49
Apres Permute : x= -49  y = 15
--------------------------------
```

Vérifions, encore une fois, ce qui se passe, lorsque l'on envoie les paramètres par valeur :

```
#include<stdio.h>
void permute(int a, int b){ // a et b reçus par valeur
    int c;
    c=a;
    a=b;
    b=c;
}
int main(){
    int x,y;
    x=15;
    y=-49;
    printf("Avant Permute : x= %d  y = %d \n",x,y);
    permute(x,y);                          //x et y envoyés par adresse
    printf("Apres Permute : x= %d  y = %d",x,y);
}
```

Exécution :

```
Avant Permute : x= 15  y = -49
Apres Permute : x= 15  y = -49
--------------------------------
```

4. *Les variables locales et les variables globales (Langage C) :*

En langage C, il y a d'autres propriétés qui caractérisent les variables :

- *Localité/Globalité de la variable* : Cette propriété indique si la variable est *globale* et donc utilisable dans *toutes les parties du programmes ou bien Locale à un sous-programme particulier* et donc utilisable dans ce *sous-programme* seulement.

Exemple :

```
#include<stdio.h>
int k=2;              // k : variable globale --> accessible partout
int max2(int x, int y){
   if(x>=y)
     return x;
   else
     return y;
}
void deux(int *x){
   int h=5;           // h variable locale ==> accessible dans la procedure deux()
   *x= *x + max2(h,k);
}
void main(){
   int y=12;
   deux(&y);
   printf("k= %d  y=%d  ", k,y);   // k accessible partout
}
```

Exécution :

```
k= 2  y=17
--------------------------------
```

- *Portée* de la variable (*Scope of a variable*): Il y a deux situations :

- *Portée de la variable dans le programme (File scope): Pour les variables déclarées en dehors des sous-programmes (variables globales), leurs portée s'étend à tout le programme.*
  Exemple : k, dans le programme précédent (block externe aux sous-programmes).
- *Pour les variables déclarées dans un sous-programme, leur portée est limitée à ce sous-programme.*
  Exemple : h, dans le programme précédent (procédure deux) et y (procédure main()).

- *Portée de variables dans les Blocs d'instructions (Block Scope) :* Cette propriété indique tous les '*Blocs d'instructions*' (à l'intérieur du *sous-programme de déclaration de la variable*) où la variable est *visible* et donc *utilisable*.
  Remarque : En langage C, les '*Blocs d'instructions*' sont créés à l'intérieur d'un même sous-programme. Chaque bloc est délimité par { et }. Voici alors deux Blocs imbriqués, déclarés dans un sous-programme SP, ainsi :

```
Sous-programme SP()
    {   int a ;        // Bloc 1 : a est visible ici : Scope de a : Bloc 1 et Bloc 2
            {
                int b ;       // Scope de b : Bloc 2
                    // Bloc 2 :   a et b sont visibles ici
            }
        // a est visible ici, mais pas b.
    }
```

Nous résumons alors :

Définitions :

i. *Variable Globale* : Une variable est dite *globale* lorsqu'elle est définie en dehors de tout sous-programme, y compris la procédure principale *main*().
La portée (*Scope*) d'une variable *globale* est alors tout le programme.

ii. *Variable Locale* : Une variable est dite *Locale* lorsqu'elle est définie à l'intérieur d'un *sous-programme* ou un *Bloc d'instructions d'un sous-programme*.

iii. La *portée* (*scope*) d'une variable *Locale* est alors respectivement : Le sous-programme où elle a été définie ou le *Bloc d'instructions où elle a été déclarée et* ses sous-Blocs.

Exemple détaillé :

```
// Variables globales – Variables locales – portée
// Durée de vie d'une variable-Variables statique – variables dynamique
#include <stdio.h>
int main()
{                                   //  Block 1
   int a=102, b=-312;                                  // a,b: variables globales → statiques (tous les blocs)
     {                                 // Block 2
              int x = 10, y = 20;                       // x, y: locales → dynamiques (blocs 2, 3 et 4)
              {                                            // Block 3
                        int g=-15, h=19;                  // g et h: locales → dynamiques (blocs 3 et 4)
                        {                           //      Block 4
                          int s=89, t=75;           // s et t :   locales → dynamiques (bloc 4)
                                  x++;                        // x accessible ici
                                  y++;                        // y accessible ici aussi
                                  printf("Block 4: s = %d  t= %d \n",s,t);       // s et t accessibles ici
                                  printf(" Block 4: g=%d  h=%d \n",g,h);        // g et h accessibles ici
                                  printf(" Block 4: x = %d, y = %d \n", x, y);  // x et y accessibles ici
                                  printf(" Block 4: a= %d   b= %d \n",a,b);    // a et b accessibles ici
                        }                                  // fin du Block 4   et durée de vie de s et t
                        printf(" Block 3: g=%d  h=%d \n",g,h);          // g et h accessibles ici
                        printf(" Block 3:  x = %d, y = %d \n", x, y);     // x et y accessible du Block 3
                        printf(" Block 3:  a = %d, b = %d \n", a, b);     // a et b accessible du Block 3
              }                                            // fin du block 3 et durée de vie de g et h
        printf(" Block 2 : x = %d  y = %d \n", x, y);           // x et y accessibles ici
        printf("Block 2: a= %d   b= %d \n",a,b);                // a et b accessibles ici
              }                                          // fin du block 2 et durée de vie de x et y
     printf("Block 1: a= %d   b= %d \n",a,b);                 // Block 1 : seuls a,b accessibles d'ici
     return 0;
}                                   // fin du block 1 et durée de vie a et b
```

Exécution :

```
Block 4: s = 89  t= 75
 Block 4: g=-15  h=19
 Block 4: x = 11, y = 21
 Block 4: a= 102   b= -312
 Block 3: g=-15  h=19
 Block 3:  x = 11, y = 21
 Block 3:  a = 102, b = -312
 Block 2 : x = 11  y = 21
Block 2: a= 102   b= -312
Block 1: a= 102   b= -312
```

Exemple 2 : Nous reprenons l'exemple de permutation de a et b, comme suit :

```
#include<stdio.h>
 int c;                          // c : variable globale (statique)
                                 //Scope: tout le programme
                                 // durée de vie: tout le programme
void permute(int *a, int *b){
    // c est accessible partout
        // a et b sont des variables locales à permute (dynamiques)
        //scope : procédure permute()
        // durée de vie : procédure permute()
    c=*a;          // c :accessible partout(globale)
    *a=*b;
    *b=c;
}
int main(){
    int x,y;
    x=15;
    y=-49;
    printf("Avant Permute : x= %d  y = %d \n",x,y);
    permute(&x,&y);
    printf("Apres Permute : x= %d  y = %d \n",x,y);
   printf("c= %d",c) ;              // c :accessible partout(globale)
}
```

Exécution :

```
Avant Permute : x= 15  y = -49
Apres Permute : x= -49  y = 15
c= 15
--------------------------------
```


iv. Usage de Blocs d'instructions : *pourquoi* ?

Le recours à l'usage des blocs d'instructions est justifié par l'argument de meilleure gestion de la mémoire réservée par les programmes. En effet, grâce aux notions de localité, aspect dynamique et portée des variables, une variable locale ne dure que pendant les moments de son utilisation dans les blocs où elle est accessible. En dehors, la variable n'existe pas : son espace mémoire est récupéré.

5. *La récursivité (programmes itératifs et programmes récursifs)*

i. Introduction :

Depuis les notions présentées dans la matière ASD1 et jusque-là, nous programmions selon les structures algorithmique de base :

- Séquence naturelle
- Structures d'alternatives
- Structures de répétitions

Ces structures algorithmiques étaient suffisantes pour proposer des solutions aux problèmes traités jusque-là. Tous les programmes réalisés jusque-là sont de nature : *itérative (itération)*. Ils relèvent de la *programmation itérative*.

Cependant, certains problèmes exigent ou peuvent mieux être résolus par un autre outil de la programmation : *La récursivité*. Au fait, la récursivité est une notion connue en mathématiques, bien avant l'avènement de l'informatique. Cette notion réfère aux fonctions qui s'appellent-elles mêmes, dans leurs définitions, pour réaliser des calculs répétitifs.

Pour illustrer ces deux notions : programmes/algorithmes *itératifs et* programmes/algorithmes *récursifs*, nous considérons un exemple de fonction mathématique, déjà traité en ASD1 : Le problème de la factorielle d'un nombre entier positif.

- Définition : $n!= 1*2*\ldots n, \quad n \geq 0,$ (I)
  Au fait, il existe une autre façon mathématique de définir n!:

$$n!=\begin{cases} 1 \ si\ n == 0\ ou\ 1 \\ n*(n-1)!\ , si\ n > 1 \end{cases} \qquad \text{(II)}$$

La solution (I) est dite *itérative* : La fonction n! est réalisée par une répétition explicite de l'opération de multiplication.

La solution (II) est dite *récursive* : La fonction n! est réalisée par une répétition implicite par appels successif de la fonction n! à elle-même.

ii. Définitions :

a. *Programme/algorithme itératif* :

Un programme est dit itératif lorsque les opérations de répétitions qu'il contient sont réalisées explicitement par les structures algorithmiques de répétition :

Pour…finpour, Tantque…FinTantque et Répéter … jusqu'à.

Exemple : Algorithme (*Fonction*) itératif pour la fonction n! :

```
fonction FactIteratif (n : entier) : entier
variables i , f : entier
 début
    f ← 1
    si (n >1) alors
        pour (i=2 jusqu'à n) faire
            f ← f*i
            i ← i+1
    finsi
   FactIteratif ← f
Fin
```

Un algorithme principal qui utilise cette fonction :

```
Algorithme Factorielle1
Données n : entier
Résultats n!: entier
Variables
Début
   Lire(n)
   Ecrire('Factorielle n =',FactIteratif(n))
Fin
```


Un programme complet et équivalent en langage C :

```
#include<stdio.h>
int FactIteratif(int n){
int i , f ;
f = 1;
if (n>1)
        for (i=2; i<=n; i++){
        f=f*i;
        }
return f;
}
int main(){
int n;
printf("Donnez n = ");
scanf("%d",&n);
printf("Factorielle %d = %d",n,FactIteratif(n));
}
```

Exécution :

```
Donnez n = 8
Factorielle 8 = 40320
--------------------------------
```

b. *Programme/Algorithme récursif :*

Un Programme/algorithme ou sous-programme (Fonction ou Procédure) est dit récursif s'il s'appelle lui-même pour réaliser une tâche donnée.

Remarquons qu'un tel programme utilise aussi les autres structures algorithmiques itératives.

Mais, il suffit d'un appel du programme (fonction ou procédure) à lui-même pour que ce programme soit considéré comme récursif.

Exemple :

Nous réalisons maintenant une solution récursive pour la fonction n!

```
Fonction FactRecursif(n : entier) : entier
Variables f : entier
Début
   Si(n>1) alors
        f ← n* FactRecursif (n-1)       // appel récursif
   Sinon
        f ←1
   Finsi
Fin
```

Un algorithme principal qui utilise cette fonction :

```
Algorithme Factorielle2
Données n : entier
Résultats n ! : entier
Variables
Début
   Lire(n)
   Ecrire('Factorielle n =',Factrecursif(n))
Fin
```

Un programme complet et équivalent en langage C :

```
#include<stdio.h>
int FactRecursif(int n){
int f ;
if (n>1)
      f=n*FactRecursif(n-1);
else
   f=1;
return f;
}
int main(){
int n;
printf("Donnez n = ");
scanf("%d",&n);
printf("Factorielle %d = %d",n,FactRecursif(n));
}
```

Exécution :

```
Donnez n = 8
Factorielle 8 = 40320
--------------------------------
```

iii. Procédures récursives :

Nous avons vu dans ce cours un exemple de fonctions itératives et fonctions récursives équivalentes.

Dans cette section, nous considérons des exemples de procédures récursives.

Exemple 1 :

Soit n un nombre entier, écrire alors une fonction récursive qui affiche tous les nombres de 1 à n,

ainsi :

Ligne 1 : 1
Ligne 2 : 1 2
…
Ligne n : 1 2 3 … n

Nous écrivons pour cela la procédure récursive suivante :

```
Procedure LignesRec(i :entier, m :entier) // Procédure Récursive
Var j : entier
    Début
      Si (i<= m) alors
         For(j=1 Jusqu'à i) faire
           Ecrire(j)
         Finpour
         LignesRec(i+1,m)      // appel récursif
      Finsi
Fin
```


Un programme principal associée est comme suit:

```
Algorithme Lignes
Données n : entier
Résultats : Ecriture de tous les nombres 1, 1..2, 1..3, …, 1..n ; ligne par ligne
Variables
Début
     Lire(n)
     LignesRec(1,n)                    // Appel initial
Fin
```

Programme équivalent en langage C :

```
#include <stdio.h>
void LignesRec(int i, int m){    // procedure récursive
 int j;
 if(i<=m){
  for(j=1;j<=i;j++)
        printf(" %d ",j);
        printf("\n");
  LignesRec(i+1,m);              // appel récursif
  }
 }
 int main(){
        int n;
        printf("Donnez n = ");
        scanf("%d",&n);
        LignesRec(1,n);          // appel initial
 }
```

Exécution :

```
Donnez n = 8
 1
 1  2
 1  2  3
 1  2  3  4
 1  2  3  4  5
 1  2  3  4  5  6
 1  2  3  4  5  6  7
 1  2  3  4  5  6  7  8
```