الجمهورية الجزائرية الديمقراطية الشعبية

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT
SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
Université Mustapha Stambouli de Mascara

Faculté des Sciences exacts
Département Informatique

وزارة التعليم
العالي و البحث العلمي
جامعة مصطفى اسطنبولي معسكر

كلية العلوم الدقيقة
قسم الإعلام الآلي

Cours

Algorithmique et Structures de Données

Cours distingué aux étudiants :

2ème année Licence informatique – 3ème semestre –

Dr. TEGGAR Hamza

Sommaire

Chapitre 1

Les éléments de base d'un algorithme

Objectif

Dans ce premier chapitre, nous allons tenter de comprendre la notion d'algorithme à l'aide de nombreux exemples dont l'objectif est de :

- Faire un rappel sur la gestion des entrées et sorties en langage C.
- Savoir comment utiliser les structures de données.
- Savoir comment décomposer les problèmes complexes.
- Apprendre à écrire les fonctions et les procédures récursives.

1.1 Définition d'un algorithme

Un algorithme est une suite d'action à exécuter par un automate (ordinateur) pour résoudre un problème particulier. A partir d'une situation donnée, que nous définirons par **précondition**, un algorithme permet une description non ambiguë, en un ensemble fini d'opérations élémentaires appelées **instructions**, qui devra être effectué par l'automate.

Ces instructions doivent être définies d'une façon rigoureuse et effective, c'est-à-dire les opérations élémentaires sont réalisées par une machine en respectant une chronologie imposée. De plus, les algorithmes que nous allons considérer dans ce cours sont déterministes. Autrement dit, toute exécution de ces algorithmes avec les mêmes données et les mêmes préconditions donne lieu à la même suite d'opérations.

1.2 Méthodologie

Pour arriver à écrire correctement un algorithme, nous devrons d'abord améliorer nos habitudes de réflexion et d'analyse des problèmes que nous souhaitons les résoudre. Dans ce qui suit, nous présenterons quelques notions de base à saisir pour bien comprendre la structure globale d'un algorithme.

1.2.1 Lisibilité

Une bonne mise en forme d'un algorithme permet de faciliter la compréhension d'un algorithme. Un bon choix de cette mise en forme (exemple 2) consiste à écrire chaque instruction élémentaire sur une seule ligne :

Exemple 1

Exemple d'une mauvaise mise en page

```
int i,n,f ;f=1;  for (i=n ; i>0 ; i--){
f=f*i;



}
```

Exemple 2

```
int i,n,f ;
f=1;
for (i=n ; i>0 ; i--){
   f=f*i;


}
```

1.2.2 Spécification d'un algorithme

La spécification nous donne une indication textuelle sur ce qu'il doit faire un algorithme, ainsi les résultats attendus de celui-ci. Dans ce cours, les spécifications sont écrites au début de l'algorithme en *Italic* entre les deux délimiteurs /∗ et ∗/.

Exemple 3

```
/* Cet algorithme calcule la factorielle d'un nombre entier positif */
     int i,n,f ;
     f=1;
     for (i=n ; i>0 ; i--){
              f=f*i;
     }
```

Exemple 4

```
/* Cet algorithme recherche l'indice de la valeur minimale dans un tableau  de
n élément  */
      n=5 ;
      int imin= 0;
      for (int i=1 ; i<n ; i++){
         if (T[i]<T[imin])
                     imin=i ;
         }
```

La spécification doit être aussi précise car la comparaison entre les algorithmes n'a de sens que si leurs spécifications sont les mêmes, ou divergent sur des cas bien répertoriés.

Dans l'exemple 4, on peut remarquer que la spécification est imprécise. La spécification doit indiquer, dans tous les cas possibles :

- **Les préconditions** : est-ce que le tableau peut être vide ?
- **Les valeurs des variables d'entrées** et éventuellement de sorties : que représentent les deux variables n et $imin$ *?*
- **Les traitements ou/et les résultats attendus dans certains cas** : qu'elles la valeur de la variable $imin$ dans le cas où il y a plusieurs minima dans le tableau. Ou dans un autre cas, qu'elle est sa valeur si le tableau est vide.

On peut aussi décrire formellement une spécification d'un algorithme. C'est-à-dire on peut utiliser des formules mathématiques ou logiques pour décrire formellement une spécification. Par exemple, une spécification formelle de l'algorithme précédent peut être décrite comme suit :

$$card\ (L) = n\ , n > 0$$

$$card\ (L) = 1 \Longrightarrow\ imin = 0$$

$$\exists k \in [0, n-1], \forall i \in [0, k-1]\ et\ \forall j \in [k, n-1], T[i] > T[k] et\ T[j] \geq T\ [k] \Longrightarrow imin = k$$

1.2.3 Les variables d'entrées et de sorties

Les valeurs des variables d'entrées sont indispensables pour faire un traitement ou un calcul. Leurs valeurs peuvent être initialisées par l'instruction d'affectation (exemple $n = 5$), ou elles peuvent être introduites par l'utilisateur lors de l'exécution du programme. Dans ce dernier cas, la fonction $scanf\ ()$ permet de lire la valeur d'une variable à partir du clavier.

Syntaxe

$$\boldsymbol{scanf}\ (<"\ format\ ">, <\&variable>)\ ;$$

- $< "\ format\ " >$: un format commence toujours par le symbole % et il est suivi par une ou plusieurs lettres qui indiquent le type de données que nous souhaitons lire. Un format doit être aussi délimité par les " " (voir le tableau);
- $< \&variable >$: la fonction $scanf()$ a besoin aussi de connaître l'emplacement en mémoire de la variable à lire. Pour effectuer cette opération, nous utilisons le symbole &, qui est en fait un opérateur que nous allons le détail plus tard.

Pour que le programme puisse interagir avec l'utilisateur du programme, la fonction $printf\ ()$ permet d'afficher des textes sur l'écran. Ces textes sont des chaînes de caractères contenant éventuellement des formats d'une ou plusieurs valeurs de variables à afficher.

Syntaxe

$$\boldsymbol{printf}\ (< "chaîne\ de\ caractères\ < ft_1 >..< ft_2 >..< ft_n >.." >, < v_1 >, \dots, < v_n >)\ ;$$

Les formats $ft_1, ft_2, \dots, ft_n$ sont des repères dans la chaîne qui indique à la fonction $printf()$ qu'une valeur d'une ou plusieurs variables $v1, v2, \dots, vn$ est attendue à cet endroit. La fonction $printf()$ doit donc contenir autant de formats ft_n que des variables v_n.

Type de variable	scanf	printf	Format

`int i;`
`int i;`
`int i;`
`unsigned int i;`
	`scanf("%d",&i) ;`
	`scanf("%o",&i) ;`
	`scanf("%x",&i) ;`
	`scanf("%u",&i) ;`
		`printf("%d",i) ;`
		`printf("%o",i) ;`
		`printf("%x",i) ;`
		`printf("%u",i) ;`
			Decimal
			Octal
			Hexadecimal
			Decimal

`short i;`
`short i;`
`short i;`
`unsigned short i;`
	`scanf("%hd",&i) ;`
	`scanf("%ho",&i) ;`
	`scanf("%hx",&i) ;`
	`scanf("%hu",&i) ;`
		`printf("%hd",i) ;`
		`printf("%ho",i) ;`
		`printf("%hx",i) ;`
		`printf("%hu",i) ;`
			Decimal
			Octal
			Hexadecimal
			Decimal

`long i;`
`long i;`
`long i;`
`unsigned long i;`
	`scanf("%ld",&i) ;`
	`scanf("%lo",&i) ;`
	`scanf("%lx",&i) ;`
	`scanf("%lu",&i) ;`
		`printf("%ld",i) ;`
		`printf("%lo",i) ;`
		`printf("%lx",i) ;`
		`printf("%lu",i) ;`
			Decimal
			Octal
			Hexadecimal
			Decimal

`float r;` `float r;`	`scanf("%f",&r) ;` `scanf("%e",&r) ;`	`printf("%f",r) ;` `printf("%e",r) ;`	**Point Decimal** **Exponentielle**
`double d;` `double d;`	`scanf("%lf",&d) ;` `scanf("%le",&d) ;`	`printf("%lf",d) ;` `printf("%le",d) ;`	**Point Decimal** **Exponentielle**
`long double d;` `long double d;`	`scanf("%Lf",&d) ;` `scanf("%Le",&d) ;`	`printf("%Lf",d) ;` `printf("%Le",d) ;`	**Point Decimal** **Exponentielle**

`char c;`
`char mot[5];`
	`scanf("%c",&c) ;`
	`scanf("%s",mot) ;`
	`scanf("%s",&mot[0]);`
		`printf("%c",c) ;`
		`printf("%s",mot) ;`
		`printf("%s",&mot[0]);`
			Caractère
			Chaîne de caractères

Exemple 5

Une spécification plus précise de l'exemple 4 serait :

```
/* les données d'entrées est T et n. T est un tableau de n éléments entiers
saisis par l'utilisateur. Le tableau ne peut pas être vide (n>0). L'indice du
premier élément dans le tableau est 0 (i=0). On cherche l'indice imin d'un élé-
ment T[i] qui représente la valeur minimale dans le tableau. L'algorithme ter-
mine avec la variable imin égale à l'indice de la première valeur minimale ap-
paru dans le tableau T * /

int n ;
int T[100] ;
int i;
printf("entrer la taille effective du tableau :") ;
scanf("%d",&n) ;
if(n>0 && n<100){
   for (i=0 ; i<n ; i++){
         printf ("enter la valeur T[%d]:",i) ;
         scanf ("%d", &T[i]) ;
   }
   int imin= 0;
   for (i=1 ; i<=n ; i++){
         if (T[i]<T[imin])
                     imin=i ;
   }
printf ("la première position du minimum dans le tableau est :%d ", imin);
```

1.2.4 Déclaration des variables et définition de nouveaux types

La spécification est généralement suivie d'une ou plusieurs déclarations des variables. Ces déclarations servent à indiquer à l'ordinateur les identificateurs de variables utilisés, ainsi que leur type. Dans ce cours nous allons choisir le langage C pour codifier nos algorithmes.

Syntaxe

Une variable est déclarée comme suit :

$$< type\ de\ variable > < identifient > \ ;$$

Exemple 6

```
int x ;    // déclaration d'une variable x de type entier
double y ; // déclaration d'une variable y de type réel
char c ;   // déclaration d'une variable c de type caractère
```

Dans certains cas, créer de nouveaux types de variables devient indispensable quand on cherche à faire des algorithmes plus complexes. Une définition de type permet, comme son nom l'indique, de définir un type, c'est-à-dire d'en produire un nouveau ou, plus précisément, de créer un alias d'un type existant. Une définition de type est identique à une déclaration de variable, si ce n'est que celle-ci doit être précédée du mot-clé ***typedef*** et que l'identificateur ainsi choisi désignera un type et non une variable.

Exemple 7

```
typedef  int entier ;            // création d'un alia de type int
typedef  char chaineMax15 [16];  // création d'un type représentant une
                                  // chaîne de caractère contenant au
                                  // maximum 15 caractères

entier x ;          // déclaration d'un entier x
chaineMax15 nom ; // déclaration d'un chaîne de caractère nom
```

1.2.5 Pointeurs

Un pointeur est une variable dont la valeur est une adresse mémoire. Pour bien comprendre la notion des pointeurs, nous devrons d'abord savoir comment le compilateur fait le lien entre les identificateurs des variables et leurs adresses mémoire.

Exemple 8

```
int x ;    // déclaration d'une variable x de type entier
int y ;    // déclaration d'une variable y de type entier
x=10 ;     // affectation de la valeur 10 à la variable x
y=5 ;      // affectation de la valeur 5 à la variable y
```

Lors de la déclaration des deux variables x et y (1ère et 2ème instruction), le compilateur réserve deux zones mémoire pour stocker leurs valeurs. Ces deux zones sont référenciées par une adresse mémoire notée respectivement $\&x$ et $\&y$. Le compilateur va utiliser donc ces deux adresses pour affecter les valeurs 10 et 5 dans les deux zones mémoire qui correspondent aux deux variables x et y (3ème et 4ème instruction).

Identificateur	adresse	valeur
x	&x	10
y	&y	5

Pour déclarer un pointeur, il faut spécifier le type de la valeur sur laquelle il va pointer comme suit :

Syntaxe

$$< \textbf{type de la valeur pointée} > * < identifient\ du\ pointeur\ >$$

Pour affecter une adresse d'une variable à un pointeur, on utilise l'opérateur &.

Syntaxe

$$< Identificateur\ d'un\ pointeur > = \ \& \ < identificateur\ d'une\ variable >$$

Et pour accéder à la valeur sur laquelle pointe un pointeur, on utilise l'opérateur *unaire d'indirection* « * ».

Syntaxe

$$* < Identificateur\ d'un\ pointeur >$$

Exemple 9

```
int x ;     // déclaration d'une variable x de type entier
int y ;     // déclaration d'une variable y de type entier
int * P ;   // déclaration d'un pointeur y qui point vers un entier
x=10 ;      // affectation de la valeur 10 à la variable x
y=5 ;       // affectation de la valeur 5 à la variable y
P=&x ;      // affectation de la adresse de x au pointeur P
printf ("P pointe sur la valeur de la variable x qui vaut: %d \n", *P);
```

Les valeurs des variables x, y et P sont présentées dans le tableau ci-dessus :

Identificateur	adresse	valeur
x	&x	10
y	&y	5
P	&P	&x

On remarque ici que P point sur la variable x, le résultat d'affichage de la fonction $printf\ ()$ sur l'écran sera :

P pointe sur la valeur de la variable x qui vaut: 10

On suppose maintenant que, après l'instruction d'affichage, on souhaite modifier la valeur de x en lui ajoutant la valeur de y :

```
int x ;      // déclaration d'une variable x de type entier
int y ;      // déclaration d'une variable y de type entier
int * P ;    // déclaration d'un pointeur y qui point vers un entier
x=10 ;       // affectation de la valeur 10 à la variable x
y=5 ;        // affectation de la valeur 5 à la variable y
P=&x ;       // affectation de la adresse de x au pointeur P
printf ("P pointe sur la valeur de la variable x qui vaut: %d \n", *P);
x=x+y ;      // modification de la variable x
printf ("P pointe sur la valeur de la variable x qui vaut maintenant: %d \n", *P);
```

Identificateur	adresse	valeur
x	&x	15
y	&y	5
P	&P	&x

Le résultat d'affichage est :

```
P pointe sur la valeur de la variable x qui vaut: 10
P pointe sur la valeur de la variable x qui vaut maintenant : 15
```

Puisque le pointeur P pointe toujours sur la valeur de x, le résultat de la deuxième instruction d'affichage donne la valeur 15.

1.2.6 Utilisation des structures de données

Le principe de base d'une structure de données, c'est de stocker des éléments sous forme d'un regroupement de variables qui peuvent avoir différents types. Pour la déclaration d'une structure de données, nous utilisons la syntaxe suivante :

Syntaxe

$$\begin{array}{l} \textbf{\textit{struct}} \ < nom\_structure > \{ \\ < Liste\ des\ sous\ variables > \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \}; \end{array}$$

Exemple 10

Imaginons par exemple que vous vouliez créer une variable qui stocke les informations d'un joueur dans un match de football. Vous aurez sans doute besoin d'une structure de données qui stocke son nom, son prénom, son équipe, et le numéro de son maillot, etc.

La structure de données permettant de stocker ces informations est déclarée comme suit :

```
struct joueur {
         chaineMax15 nom ;
         chaineMax15 prenom ;
         int numeroMaillot ;
  };
```

Pour déclarer une variable de type de la structure définie, on utilise la syntaxe suivante:

Syntaxe

$$\textbf{\textit{struct}} \ < nom\ de\ la\ structure\ définie > < identifient >;$$

et pour accéder à une l'une des sous-variables de la structure de données :

$$< identifient > . < sous - variable >$$

Exemple 11

```
struct joueur j1;          // déclaration d'une variable j1 de type structure joueur
strcpy (j1.nom, "Mahrez"); // copie le nom du joueur dans la sous variable nom
strcpy (j1.prenom,"riyad");// copie le prénom du joueur dans la sous variable prénom
j1.numeroMaillot=7;        // affecter le numéro 7 à la sous variable numeroMaillot

      printf ("joueur Numero: %d\n \n",j1.numeroMaillot);
      printf ("nom: %s\n",j1.nom );
      printf ("prenom: %s\n",j1.prenom );
```

1.2.7 Les procédures

Lorsque le problème concret est complexe, il devient nécessaire de le décomposer en sous-problèmes plus simples, plus ou moins indépendants. Par exemple, on veut faire la fusion de deux tableaux non triés V1 et V2 dans un tableau V3, tel que V3 résultant doit être trié dans un ordre croissant. Pour réaliser cette opération, on peut suivre les étapes suivantes :

1. Trier V1
2. Trier V2
3. Fusionner V1 et V2 dans V3

On remarque qu'un même problème a été résolu deux fois (le tri de vecteur). Donc le but principal visé par la définition d'une procédure est de pailler à cet inconvénient.

Une procédure est une suite d'instructions nommée, que nous pouvons appeler par ce nom chaque fois que nous voulons l'utiliser. La suite d'instructions représente l'action ou un traitement à faire par la procédure.

Une procédure est déclarée selon la syntaxe suivante :

Syntaxe

$$< \textit{déclaration de variables globales} >$$

$$\boldsymbol{void} < \boldsymbol{nomProcedure} > (< \textit{liste de paramètres} >) \{$$
$$< \textit{déclaration de variables locales} >$$
$$< \textit{corps de la procédure} >$$
$$\}$$

La première ligne est appelée en-tête, elle est composée de :

- ***void*** : contrairement à une fonction, une procédure ne renvoie aucune valeur par le billet de son nom. Le mot clé *void* (signifie vide) permet d'indiquer au programme appelant que aucune valeur n'est attendue à la fin de l'action.

- $< nomPrécedure >$: le nom de la procédure doit être une chaîne alphanumérique et qui commence obligatoirement par une lettre. Ce nom sera ensuite utilisé par une autre action appelée ***action appelante*** pour faire appel à cette procédure.

- $< Liste\ de\ paramètres >$: une liste éventuellement vide des paramètres de la procédure déclarés entre les deux parenthèses et séparées par des **« , »**. Ces paramètres sont appelés paramètres formels. Leur valeur n'est pas connue lors de la création de la procédure. Dans le cas où cette liste est vide, la procédure est appelée procédure non paramétrée.

- $< corps\ de\ la\ procédure >$: englobe les déclarations des variables locales et les instructions qui seront exécutées par la procédure. Ces instructions sont délimitées par les deux accolades { }.

- $< déclaration\ de\ variables\ locales >$: les variables locales sont des variables que ne peuvent être manipulées que par la procédure qui les déclare. Autrement dit, la portée de visibilité de ces variables ne dépasse pas le corps de la procédure où elles ont été déclarées, et aucune autre procédure ne peut les utilisées.

- $< déclaration\ de\ variables\ globales >$ **:** Contrairement aux variables **locales, les variables globales** sont déclarées hors du corps de n'importe quelle procédure dans le programme. Les variables globales sont donc visibles et elles peuvent être utilisées par toutes les procédures déclarées dans le programme principal.

Exemple 12

```
/* Cette procédure affiche le message "Bienvenue au Cours
   Algorithmique et Structures de Données" */
   void afficherMessage(){
      printf ("Bienvenue au Cours Algorithmique et Structures
                de Données !\n");
   }

/* la procédure principal main fait appel à la procédure non
   paramétrée afficherMessage() pour afficher le message */
   void main(){
      afficherMessage();
   }
```

1.2.8 Les fonctions

Les fonctions sont appelées souvent pour faire un calcul particulier. Il s'agit, comme une procédure, d'une action qui peut être appelée simplement en écrivant le nom de la fonction suivi d'une liste optionnelle de paramètres.

À l'exception d'une procédure, une fonction **doit retourner obligatoirement une seule valeur du type simple à son point d'appel**. Pour cela, la fonction doit contenir, quelque part dans son

corps, l'instruction *return* qui sera la dernière instruction à exécuter. Une fois l'instruction *return* est exécutée on quitte le corps de la fonction.

Pour la déclaration d'une fonction, il faut précéder son nom par le type de la valeur retournée. Par conséquent, lors d'un appel, une fonction est considérée comme une expression de type de la valeur retournée (exemple : si la fonction retourne un entier, on considère la fonction comme une expression entière).

Syntaxe

$< \textbf{type de la valeur retournée} > < \textbf{nomFonction} > (< \text{liste de paramètres} >) \{$
$< \text{déclaration de variables locales} >$
$< \text{corps de la procédure} >$
$\textbf{return} < \text{valeur} > ;$
$\}$

Exemple 13

```
/* Cette fonction demande l'âge de l'utilisateur, puis elle renvoie 1 s'il
   est majeur(age >18), 0 sinon */

        int majeur(){
           int age;
           printf("Enter votre age : ");
           scanf("%d",&age);
           if (age>18)
                 return 1;
           else return 0;
        }
```

Quel que soit l'âge de l'utilisateur, on remarque que la dernière instruction exécutée par la fonction sera *return*.

Exemple 14

Dans cet exemple, on souhaite demander l'âge de l'utilisateur, puis on affiche sur l'écran les messages suivants :

```
Bienvenue au Cours Algorithmique et Structures de Données !
Enter votre âge : 19
Vous êtes majeur !
```

```
/* 1e version d'un algorithme qui réalise les actions de l'exemple 11.
   Dans cette version, on fait appel à la procédure afficherMessage()
   de l'exemple 8 pour afficher la première ligne, puis on récupère dans
   la variable res le résultat de la fonction majeur() pour afficher
   la troisième ligne. La deuxième ligne sera affichée lors de l'appel
   de la fonction majeur() */
```

```
void main(){
   int res;
   afficherMessage();
   res=majeur();
   if(res==1)
         printf ("vous êtes majeur");
   else
         printf ("vous êtes mineur");
}
```

Dans cette deuxième version, on peut se débarrasser de la variable res. On peut utiliser la fonction $majeur\ ()$ comme une expression. L'instruction $if\ (res == 1)$ peut être remplacée par $if\ (majeur() == 1)$ ou plus simplement par $if(majeur())$.

```
/* 2e version sans l'utilisation de la variable res'*/

void main(){
   afficherMessage();
   if(majeur())
         printf ("vous êtes majeur");
   else
         printf ("vous êtes mineur"); }
```

1.2.9 Passage de paramètres pour les fonctions et les procédures

Un paramètre est une donnée fournie par l'action appelante lors d'un appel d'une procédure ou une fonction. Cette donnée peut être transmise de deux façons : passage par valeur ou passage par adresse (on dit aussi par référence).

Passage par valeur :

Une fonction ou une procédure utilisant ce mode de passage de paramètres, n'utilise que la valeur de l'expression passée en argument. Cette expression peut être :

- Une valeur (exemple 5, 1^{e-1}, 'a', «chaîne de caractères », etc).
- Une variable.
- Une expression arithmétique ou logique.
- Un appel d'une fonction.

Si le paramètre passé en argument est une variable locale de la « fonction appelante », la valeur de cette variable est copiée dans une autre variable locale de la « fonction appelée ». C'est cette variable qui sera utilisée pour faire les calculs dans la « fonction appelée », et cela sans aucune modification des variables de la « fonction appelante ».

Syntaxe

$< type\ de\ valeur\ retournée > < nomAction > (< type\ valeur > par1, \dots, < type\ valeur > parn)\ \{$

$\}$

Exemple 15

```
/* On veut écrire une procédure formule() qui demande les valeurs de deux entiers n
et y, puis elle calcule et stocke dans une variable locale x la valeur n!+y!. Cette
fonction fait appel à la fonction factorielle*/
```

```
int factorielle (int n){
   int f=1;
         while (n>0){
               f=f*n;
               n=n-1;}
   return f;
}

void formule(){
   int n, y, x;
   printf (" Entrer la valeur n:");
   scanf ("%d",&n);
   printf (" Entrer la valeur y:");
   scanf ("%d",&y);
   x=factorielle (n) + factorielle (y);
   printf ("%d !+ %d != %d",n,y,x);
}
```

On fait appel à la procédure $formule()$ dans la fonction principale comme suit :

```
int main (){
   formule();
   return 0;
}
```

L'exécution de ce code pour les valeurs $n = 3$ et $y = 5$ donne le résultat suivant :

```
Entrer la valeur n:3
Entrer la valeur y:5
3 !+ 5 != 126
```

La dernière instruction dans la procédure formule() affiche la dernière ligne de l'exécution. On remarque que les valeurs des deux variables locales n et y n'ont pas été modifiés par la fonction factorielle. Cette dernière n'utilise que leurs valeurs pour le calcule de la factorielle.

Passage par variable (ou par adresse, par référence)

Le deuxième mode de passage des paramètres consiste à passer non plus la valeur d'une variable comme argument, mais à passer la variable elle-même. La « fonction appelée » va donc utiliser cette variable comme si elle est déclarée localement. Toute modification par la « fonction appelée » du paramètre, passé par ce mode, entraîne la modification de la variable passée en argument par la « fonction appelante ».

Syntaxe

$$< type\ de\ valeur\ retournée > < nomAction > (< type\ valeur > * par1, \dots, < type\ valeur > * parn) \{ \}$$

Pour réaliser ce mode de passage de paramètres en langage C, Il n'y a qu'une solution : au lieu d'utiliser l'identificateur de la variable pour faire des appels, il va falloir passer l'adresse de la variable. Cela est possible grâce à utilisation de la notion des pointeurs.

Le paramètre $< type\ valeur > *\ par1$ indique qu'il s'agit d'un pointeur et non pas un type d'une valeur simple.

Exemple 16

```
/* Dans l'exemple 14,  la procédure formule calcule localement la valeur (n !+y !)
      pour la stocker ensuite dans la variable locale x. Lors de l'appel de cette
      procédure au niveau de la fonction principale main(), il est impossible de
      récupérer la valeur de la variable x.
      Dans cet exemple  nous allons modifier la procédure formule (), pour que nous
      puisse faire passer la variable x qui sera déclarée localement dans la fonc-
      tion principale au lieu de la procédure formule(). L'affichage de résultat se-
      ra aussi effectuer au niveau de la fonction principale.  */

      void formule(int * x){
         int n, y;
         printf (" Entrer la valeur n:");
         scanf ("%d",&n);
         printf (" Entrer la valeur y:");
         scanf ("%d",&y);
         *x=factorielle (n)+ factorielle (y);
      }
```

On fait appel à la procédure $formule()$ dans la fonction principale comme suit :

```
      int main (){
         int x;
          formule(&x); // on fait un appel par passage d'adresse de la variable x
         printf ("Le résultat du calcul est : %d",x);
         return 0;
      }
```

Exemple 17

```
/* Dans cet exemple, toutes les variables sont déclarées localement dans la fonction
     principale. La procédure formule va calculer uniquement la valeur (n !+y !) */

      void formule(int n, int y, int * x){
         *x=factorielle (n)+ factorielle (y);
      }
```

On fait appel à la procédure $formule()$ dans la fonction principale comme suit :

```
      int main (){
         int x, n, y;
          printf (" Entrer la valeur n:");
         scanf ("%d",&n);
         printf (" Entrer la valeur y:");
         scanf ("%d",&y);
          formule(n, y, &x);
         printf ("Le résultat du calcul est : %d",x);
         return 0;}
```

Les valeurs n et y sont déclarées et lues au niveau de la fonction principale. Puisque la procédure $formule()$ ne modifie pas leurs valeurs, on utilise donc le mode de passage par valeur uniquement pour ces deux variables.

1.2.10 Récursivité

Une action A est dite « récursive » si l'action A, lors de son exécution, fait un appel à elle-même. Les appels récursifs sont :

- **Directe** : si l'action A appelle directement A, dans ce cas on dit que la récursivité est directe.
- **Indirecte** : si A appelle $A1$, et que si $A1$ appelle $A2$, …, et que si enfin An appelle A, on dit que la récursivité est indirecte.

La récursivité permet d'écrire des actions plus lisibles, de manière très rapide par rapport d'une manière itérative, et cela en appliquant le principe de diviser pour résoudre.

Un algorithme récursif doit contenir au moins un cas trivial qui peut être résolu sans aucun appel récursif. Par exemple la $factorielle\ (0) = 1$. En effet, pour écrire correctement un algorithme récursif, les séquences de tous les appels récursifs doivent mener, d'une manière ou d'une autre, à l'un de ces cas triviaux. Autrement dit, les cas triviaux sont considérés comme des conditions terminales pour l'arrêt de la récurrence.

Exemple 18

```
/*    Cette procédure récursive affiche un palindrome à partir d'une
      chaîne de caractères "chaîne" */
          void AffichagePlindrome(char *chaine ){
          char *s=chaine;
          if (*chaine!= '\0')  {
             printf("%c ",*chaine) ;
             AffichagePlindrome (++s) ;
             printf("%c ",*chaine) ;
          }
          }
```

Dans le programme principal, pour une chaîne $s =$ « ABC », l'appel de la procédure `Affichage-Plindrome()` se fait comme suit :

```
          void main (){
             char *s="ABC";
             int n=7;
             AffichagePlindrome(s);
          }
```

La procédure récursive affichera sur l'écran :

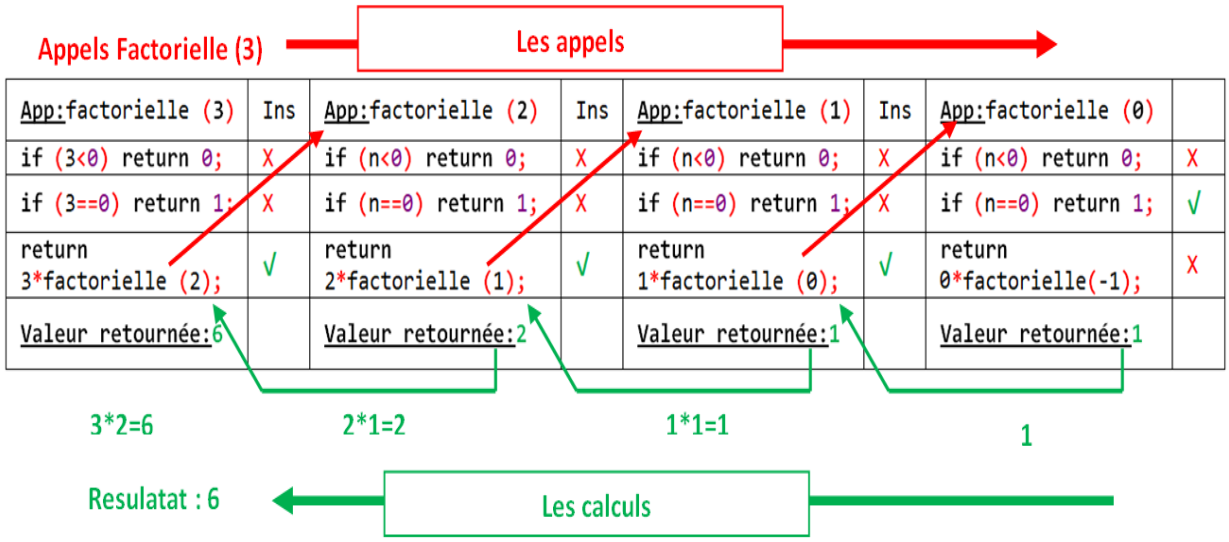
A B C C B A

Exemple 19

```
/*    version 1 : fonction récursive pour calculer la factorielle d'un nombre n */

        int factorielle (int n){
           if (n<0) return 0;
           else if (n==0) return 1;
                        else return n* factorielle (n-1);
        }


/*    version 2 : fonction récursive pour calculer la factorielle d'un nombre n */

        int factorielle (int n){
           if (n<0) return 0;
           if (n==0) return 1;
           return n* factorielle (n-1);
        }
```

Tout appel d'une fonction donne lieu à un empilement du contexte d'exécution. Ce contexte n'est rien d'autre que l'état d'exécution du programme au moment exact de l'appel d'une fonction. Les valeurs des paramètres, les valeurs des variables locales, l'adresse de l'instruction à exécuter après le retour de la « fonction appelée », etc., seront donc sauvegardées avant l'appel. Une pile d'exécution (call stack) du programme en cours d'exécution est un emplacement mémoire destiné à mémoriser ce contexte.

Dans l'exemple ci-dessous, l'appel de la fonction $factorielle(3)$ donne la séquence des appels suivante:



Exemple 20

```
/*  factorielleRecIndir() est une fonction récursive indirecte qui calcule
      la factorielle d'un entier n. Cette fonction fait appel à la fonction
      Factorielle_N_Moins_Un qui permet d'afficher la séquence de calcule
      puis cette dernière fait appel à la fonction factorielleRecIndir() */

          int factorielleRecIndir(int n){
             if(n<0) return 0;
             else if(n==0) return 1;
                      else return n* Factorielle_N_Moins_Un (n);
          }

          int Factorielle_N_Moins_Un(int n){
             if((n-1)==0) printf ("1 =");
             else
                    printf("%d * ", n);
             return factorielleRecIndir(n-1);;
          }
```

Dans le programme principal, l'instruction `printf ("%d \n", factorielleRecIndir(7));` affichera sur l'écran :

```
7 * 6 * 5 * 4 * 3 * 2 * 1 =5040
```