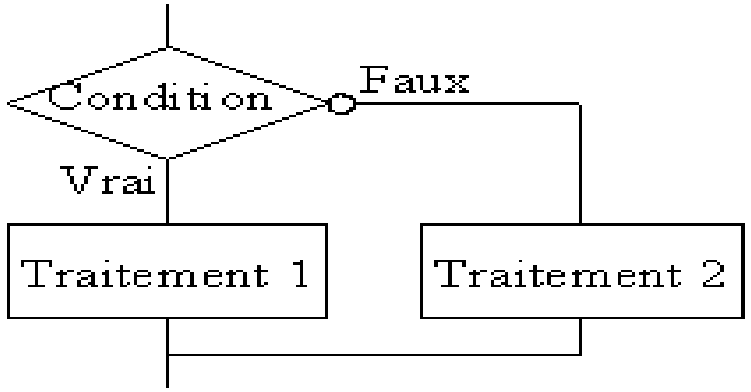
CHAPITRE 1

CONNAITRE LES BASES

1 - Le traitement alternatif (conditions)

Traitement alternatif

Rappel: Les instructions conditionnelles servent à n'exécuter une instruction ou une séquence d'instructions que si **une condition est vérifiée.**



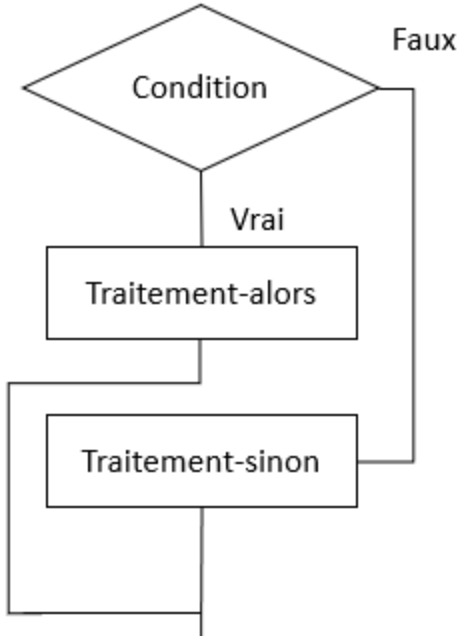
- **Alternative Si-Alors-Sinon**
- **Schéma conditionnel généralisé ou multiple**

Traitement alternatif

▪Alternative Si-Alors-Sinon

- Elle permet d'effectuer tel ou tel traitement en fonction de la valeur d'une condition

Organigramme



En pseudo-code

```
Si condition  Alors
    <Actions_alors>
Sinon
    <Actions_sinon>
Fsi
```

Traitement alternatif

Exemple 1:

```
si ( a≠0 )
alors
        a := 0
sinon
        a := b
        c := d
 fsi
```

si la condition est vraie c.à.d. la variable a est différente de 0 alors on lui affecte la valeur 0, sinon on exécute le bloc sinon.

Exemple 2:

```
si ( a – b ≠ c )
alors
        a := c
sinon
        a := d
fsi
```

si la condition est vraie, la seule instruction qui sera exécutée est l'instruction d'affectation a := c. Sinon la seule instruction qui sera exécutée est l'instruction d'affectation a := d.

Traitement alternatif

▪**Schéma conditionnel généralisé ou multiple**

La structure **cas** permet d'effectuer tel ou tel traitement en fonction de la valeur des conditions 1ou 2 ou ..n

En pseudo-code

Cas <var> de:

<valeur 1> **:** <action 1>

< valeur 2> **:** <action 2>

…

< valeur n> **:** <action n>

Sinon : <action_sinon>

FinCas

la partie action-sinon est facultative

Traitement alternatif

Exemple 1:

On dispose d'un ensemble de tâches que l'on souhaite exécuter en fonction de la valeur d'une variable choix de type entier, conformément au tableau suivant :

Valeur de choix	Tâche à exécuter
1	Commande
2	Livraison
3	Facturation
4	Règlement
5	Stock
Autre valeur	ERREUR

Traitement alternatif

▪Alternative Si-Alors-Sinon

```
si          choix = 1
alors       Commande
sinon       si   choix = 2
            alors       Livraison
            sinon       si          choix = 3
                        alors       Facturation
                        sinon       si
            choix = 4
                                    alors
            Règlement
                                    sinon       si
            choix = 5

            alors       Stock

            sinon       ecrire (''Erreur'')
            fsi
                                    fsi
                        fsi
            fsi
fsi
```

▪Structure à choix multiples

```
Cas choix de:
1:          Commande
2:          Livraison
3:          Facturation
4:    Règlement
sinon       ecrire (''Erreur'')

finCas
```

L'emboîtement de si en cascade est fastidieux à écrire

CHAPITRE 1

CONNAITRE LES BASES

2 - Le traitement itératif (boucles)

Traitement itératif

- **Instruction itérative « pour »**

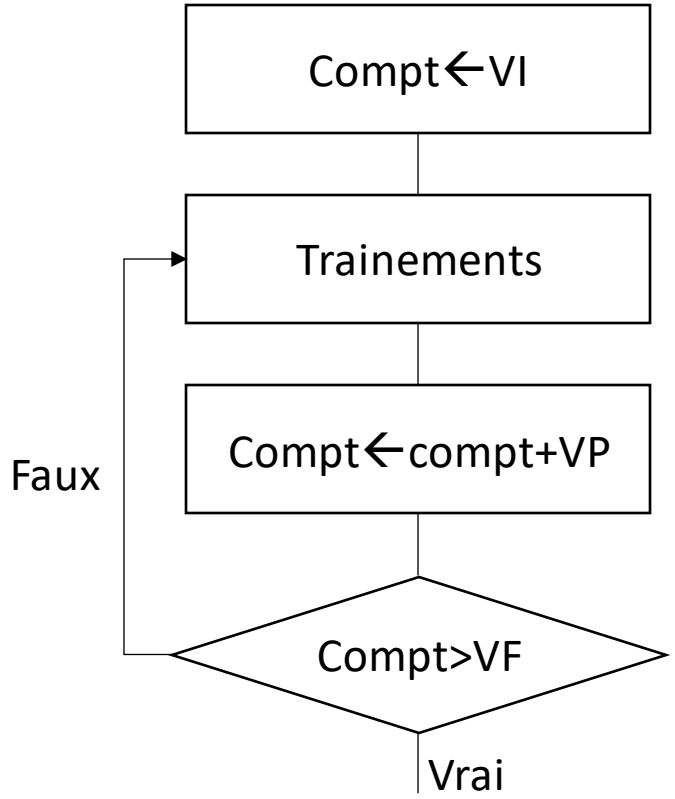
- Répéter N fois une suite d'instructions à l'aide d'une variable entière servant de compteur

Valeur initiale

Valeur finale

```
Pour <compt> de <VI> à <VF> [pas <VP>] faire
    instructions
Fpour
```

Valeur à ajouter à compt à chaque passage dans la boucle



Le traitement itératif

Exemple: un algorithme permettant de lire N entiers, de calculer et d'afficher leur moyenne

```
moyenne
var   n, i, x, s : entier
Début
        lire( n )
        s := 0
        pour  i  de  1  à  n  faire
                lire( x )
                s := s + x
        fpour
        ecrire( "la moyenne est :", s / n )
Fin
```

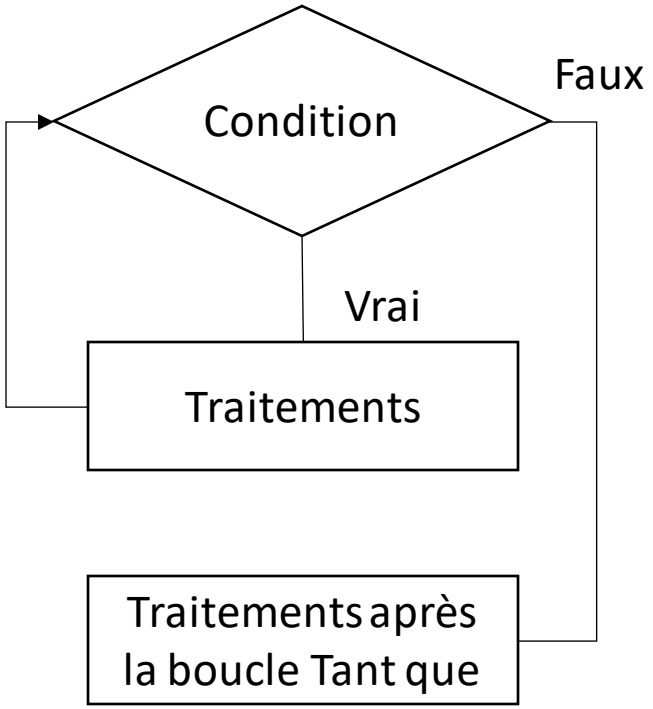
Traitement itératif

Le traitement itératif

- **Instruction itérative « Tant que »**

Répéter une suite d'instructions tant que **la condition est vraie**

```
Tant que <condition> faire
    Instructions
Fait
```



Le traitement itératif

Exemple: un algorithme permettant de lire une suite d'entiers positifs, de calculer et d'afficher leur moyenne.

```
moyenne
var    i, x, s : entier

Début

        lire( x )
        s := 0
        i := 0
        tant que  x > 0  faire
                i := i + 1
                s := s + x
                lire( x )
        fait
        si   i ≠ 0
        alors  écrire(  "la moyenne est :", s / i )
        fsi
Fin
```

Condition obligatoire pour éviter de diviser par 0 si le premier entier lu est 0

Le traitement itératif

Exemple: un algorithme permettant de lire une suite d'entiers positifs, de calculer et d'afficher leur moyenne.

```
moyenne
var    i, x, s : entier

Début

      lire( x )
      s := 0
      i := 0
      tant que  x > 0  faire
            i := i + 1
            s := s + x
            lire( x )
      fait
      si  i ≠ 0
      alors  écrire( "la moyenne  est :", s / i )
      fsi
Fin
```

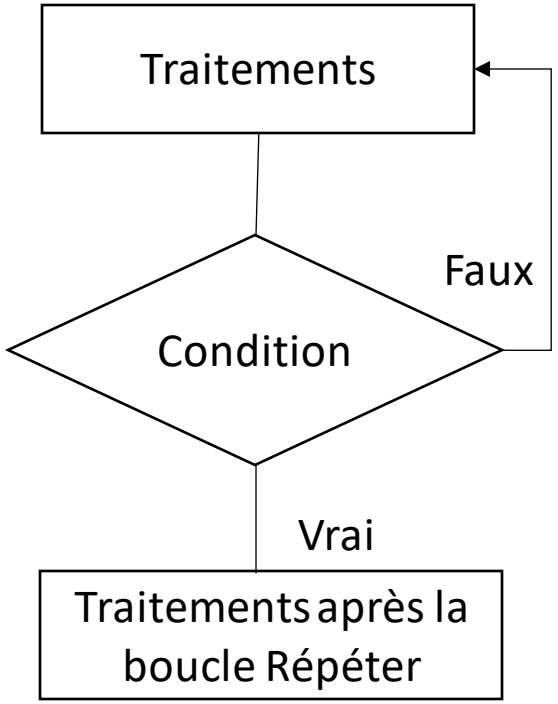
Condition obligatoire pour éviter de diviser par 0 si le premier entier lu est ≤ 0

Le traitement itératif

- **Instruction itérative « Répéter »**

Répéter une suite d'instructions jusqu'à que la condition soit évaluée à Faux

```
Répéter
        instructions
Jusqu'à <condition>
```



Le traitement itératif

Exemple: un algorithme permettant de lire deux entiers, de calculer et d'afficher le résultat de la division du premier par le second (quotient)

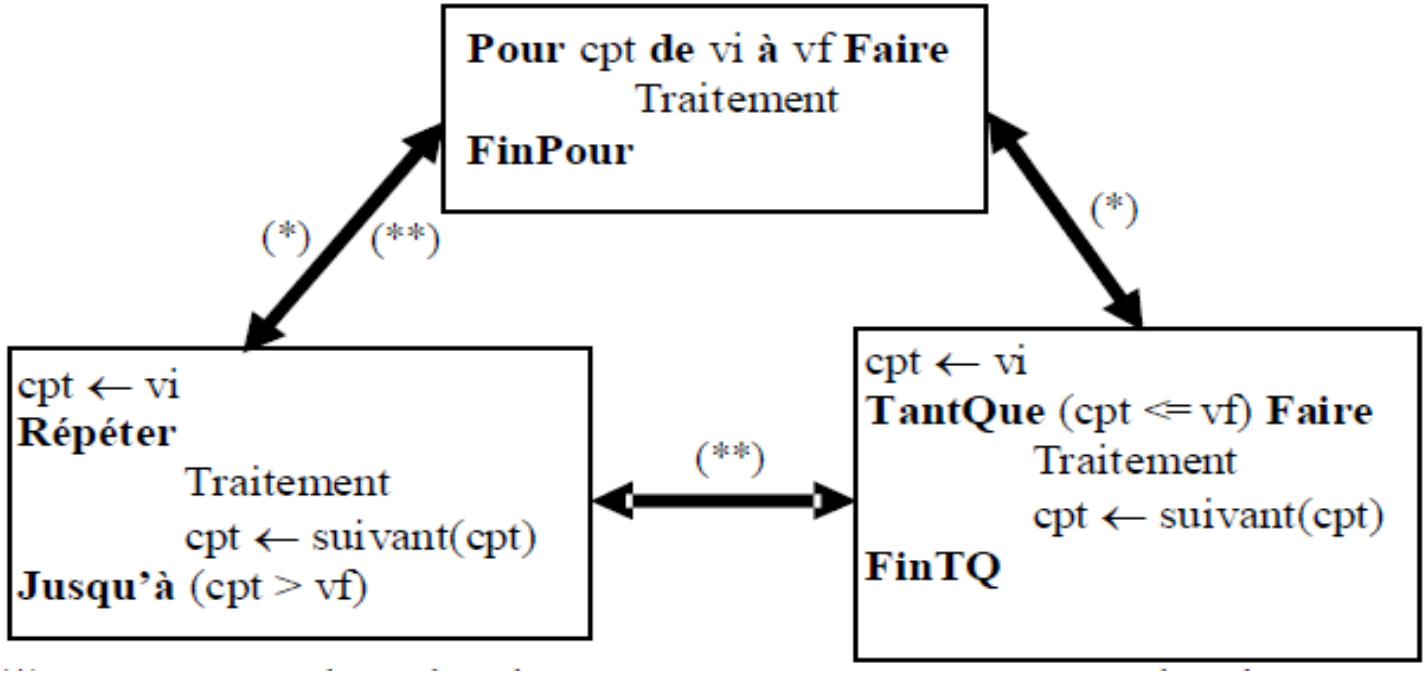
```
quotient
var   x, y : entier

Début

      lire( x )
      répéter
          lire( y )
      jusqu'à  y > 0
      écrire( x / y )
Fin
```

un contrôle obligatoire doit être effectué lors de la lecture de la deuxième valeur

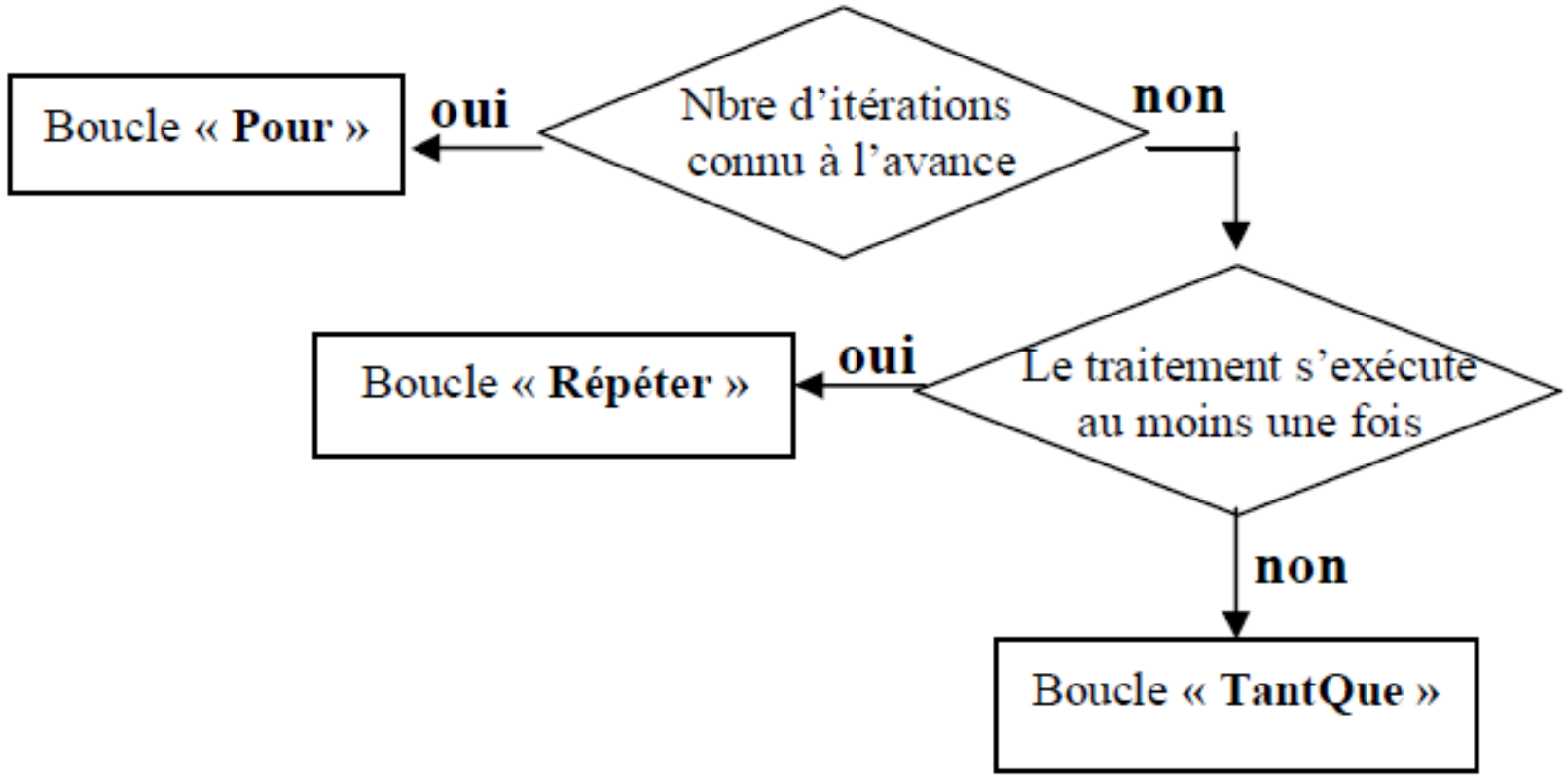
Passage d'une structure itérative à une autre



(*) : Le passage d'une boucle « répéter » ou « tantque » à une boucle « pour » n'est possible que si le nombre de parcours est connu à l'avance

(**) : Lors du passage d'une boucle « pour » ou « tantque » à une boucle « répéter », faire attention aux cas particuliers (le traitement sera toujours exécuté au moins une fois)

Choix de la structure itérative



CHAPITRE 2

SAVOIR STRUCTURER LES DONNEES

1 - Les tableaux

Structure de données

une structure de données est une manière particulière de **stocker et d'organiser des données** dans un ordinateur de façon à pouvoir être utilisées efficacement.

Une structure de données regroupe :

- Un **certain nombre de données** à gérer,
- Un ensemble d'opérations pouvant être appliquées à ces données

Dans la plupart des cas, il existe:

- Plusieurs manières de représenter les données et
- Différents algorithmes de manipulation.

Structure Tableau Vecteur

- Stocker à l'aide d'une seule variable un ensemble de valeurs de même type
- Un tableau unidimensionnel est appelé vecteur

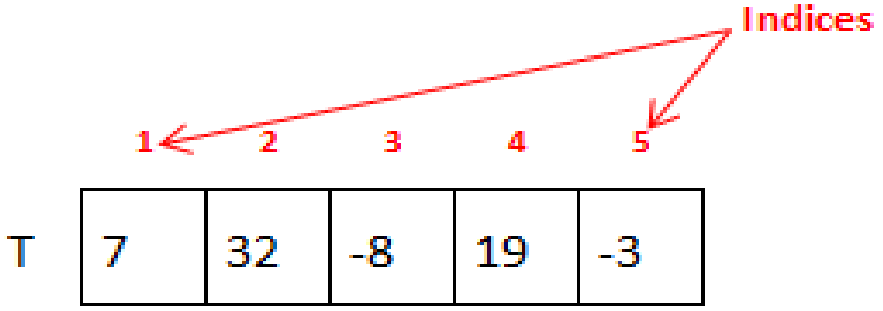
Type vecteur = tableau [1..MAX] de type des éléments

avec **MAX** est le nombre maximum d'éléments pour le type vecteur

Structure Tableau Vecteur

- Exemple de tableau de 5 cases

T : tableau [1..5] d'entier



- Accès à un élément du tableau

nom_tableau[indice]

Exemple : T[2] correspond à la case ayant la valeur 32

Structure Tableau Vecteur

▪**Caractéristiques**

- Un tableau vecteur possède un **nombre maximal d'éléments** défini lors de l'écriture de l'algorithme (les bornes sont des constantes explicites, par exemple MAX, ou implicites, par exemple 10)

- Le nombre d'éléments maximal d'un tableau est différent du nombre d'éléments significatifs dans un tableau

Exemple: un algorithme permettant de lire un tableau vecteur de 12 entiers

```
LectutreTabVecteur

Var   i : entier
      T : tableau[1..12] de  Réel
Debut
   Pour  i  de  1  à  12  faire
         Lire(T[i])
Fpour
Fin
```

Structure de tableau multi-dimensions

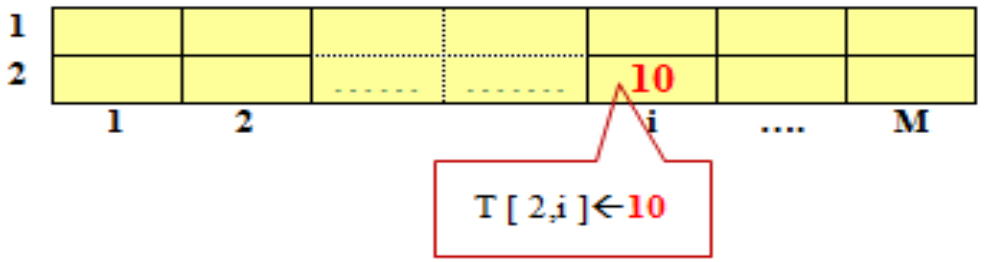
- Par extension, on peut définir et utiliser des tableaux à n dimensions

tableau [intervalle1,intervalle2,…,intervallen] de type des éléments

- Les tableaux à deux dimensions permettent de représenter les matrices

tableau [intervalle1,intervalle2] de type des éléments

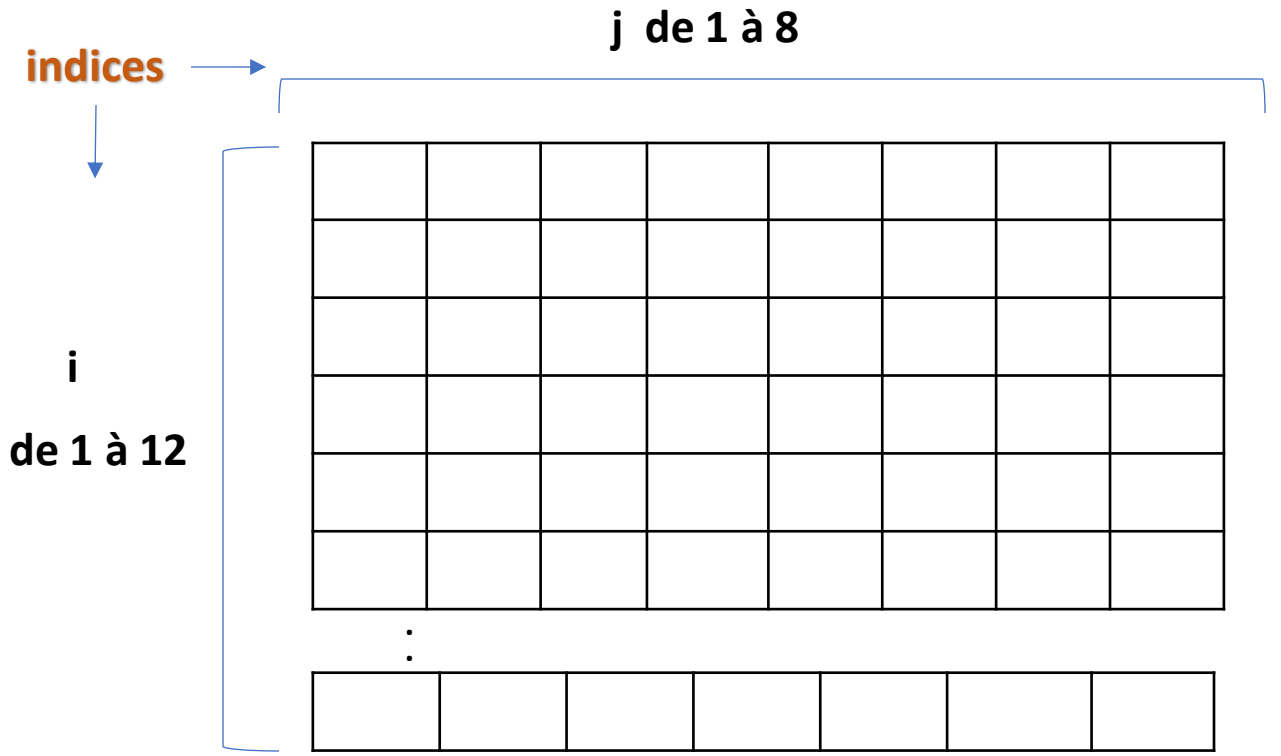
T : tableau [1..2,1..M] d'entier



Structure de tableau multi-dimensions

Exemple: un algorithme permettant de lire un tableau matrice d' entiers de 12 lignes et 8 colonne

```
LectutreTabMatrice
Var   i , j : entier
     T: tableau[1..12 , 1..8] de  Réel
Debut
   Pour i de 1 à 12 faire
        Pour j de 1 à 8 faire
            Lire(T[i , j])
        Fpour
   Fpour
Fin
```



Tri d’un tableau

Il existe plusieurs méthodes de tri qui se différencient par leur complexité d’exécution et leur complexité de compréhension pour le programmeur.

Parmi les méthodes de Tri d’un tableau on cite:

- Tri à bulle
- Tri par sélection
- Tri par insertion

Tri à bulle

Soit T un tableau de n entiers.

La méthode de tri à bulles nécessite deux étapes :

- Parcourir les éléments du tableau de 1 à (n–1) ; si l'élément i est supérieur à l'élément (i+1), alors on les permute

- Le programme s'arrête lorsqu'aucune permutation n'est réalisable après un parcours complet du tableau.

Tri à bulle

```
Procédure Tri_Bulle (Var T : Tab)
Variables
        i, x : Entier
        échange : Booléen
Début
        Répéter
                échange ← Faux
                Pour i de 1 à (n-1) Faire
                        Si (T[i] > T[i+1]) Alors
                                x ← T[i]
                                T[i] ← T[i+1]
                                T[i+1] ← x
                                échange ← Vrai
                        FinSi
                FinPour
        Jusqu'à (échange = Faux)
Fin
```

Tableau initial	6	4	3	5	2
Après la 1ère itération	4	3	5	2	6
Après la 2ème itération	3	4	2	5	6
Après la 3ème itération	3	2	4	5	6
Après la 4ème itération	2	3	4	5	6

Tri par sélection

C’est la méthode de tri la plus simple, elle consiste à :

- chercher l’indice du plus petit élément du tableau T[1..n] et permuter l’élément correspondant avec l’élément d’indice 1
- chercher l’indice du plus petit élément du tableau T[2..n] et permuter l’élément correspondant avec l’élément d’indice 2
- ……..
- chercher l’indice du plus petit élément du tableau T[n-1..n] et permuter l’élément correspondant avec l’élément d’indice (n-1).

Tri par sélection

```
Procédure Tri_Selection(Var T : Tab)
Variables
        i, j, x, indmin : Entier
Début
        Pour i de 1 à (n-1) Faire
                indmin ← i
                Pour j de (i+1) à n Faire
                        Si (T[j] < T[indmin]) Alors
                                indmin ← j
                        FinSi
                FinPour
                x ← T[i]
                T[i] ← T[indmin]
                T[indmin] ← x
        FinPour
Fin
```

Tri par insertion

Cette méthode consiste à prendre les éléments de la liste un par un et insérer chacun dans sa bonne place de façon que les éléments traités forment une sous-liste triée.

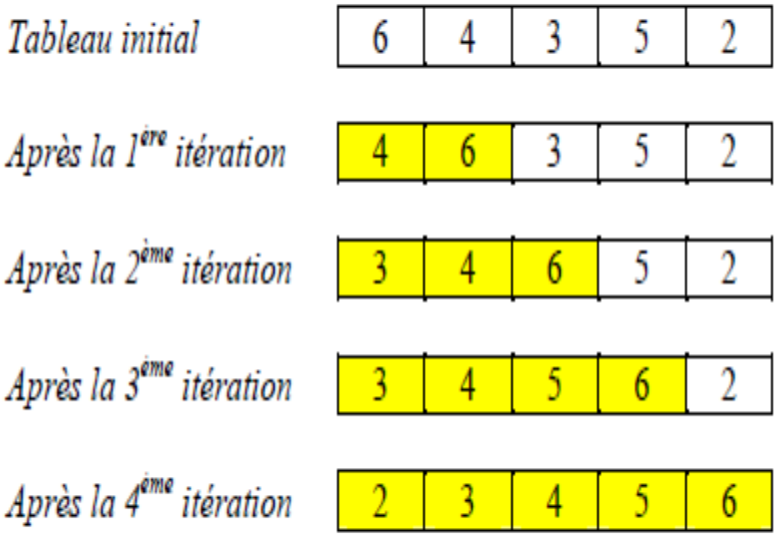
Pour ce faire, on procède de la façon suivante :

- comparer et permuter si nécessaire T[1] et T[2] de façon à placer le plus petit dans la case d'indice 1
- comparer et permuter si nécessaire l'élément T[3] avec ceux qui le précèdent dans l'ordre (T[2] puis T[1]) afin de former une sous-liste triée T[1..3]
- ……
- comparer et permuter si nécessaire l'élément T[n] avec ceux qui le précèdent dans l'ordre (T[n-1], T[n-2], …) afin d'obtenir un tableau trié.

Tri par insertion

```
Procédure Tri_Insertion(Var T : Tab)
Variables
        i, j, x, pos : Entier
Début
        Pour i de 2 à n Faire
                pos ← i - 1
                TantQue (pos>=1) et (T[pos]>T[i]) Faire
                        pos ← pos – 1
                FinTQ
                pos ← pos + 1
                x ← T[i]
                Pour j de (i-1) à pos [pas = -1] Faire
                        T[j+1] ← T[j]

                FinPour
                T[pos] ← x
        FinPour
Fin
```



[Pas = -1] signifie que le parcours se fait dans le sens décroissant

CHAPITRE 2

SAVOIR STRUCTURER LES DONNEES

2 - Les chaînes de caractères

Chaines de caractères

- Une chaine de caractères (en anglais : string) est un type structuré similaire à un tableau de caractères, et représentant une suite de caractères.
- Cette structure est prédéfinie dans la plupart des langages de programmation et associé à des fonctions

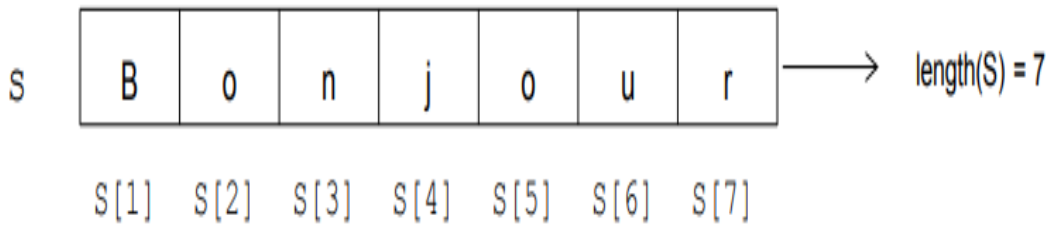
```
S : chaîne
```

- Le nombre de caractères d'une chaine est accessible via la fonction **length(<NomVar>)**

<u>Exemple:</u>

Var c:=''Saleh''

Length(c) → 5



CHAPITRE 2

SAVOIR STRUCTURER LES DONNEES

3 - Les fichiers

Fichiers

Un fichier est une structure de données formée de **cellules contiguës** permettant l'implantation d'une suite de données en mémoire secondaire (disque, disquette, CD-ROM, bande magnétique, etc.)

Chaque élément de la suite est appelé **article**

Exemples :

- liste des étudiants d'une institution
- état des produits stockés dans un magasin
- liste des employés d'une entreprise.

Eléments attachés à un fichier

On appelle **nom interne d'un ficher** le nom sous lequel un fichier est identifié dans un programme.

On appelle **nom externe d'un fichier** le nom sous lequel le fichier est identifié en mémoire secondaire.

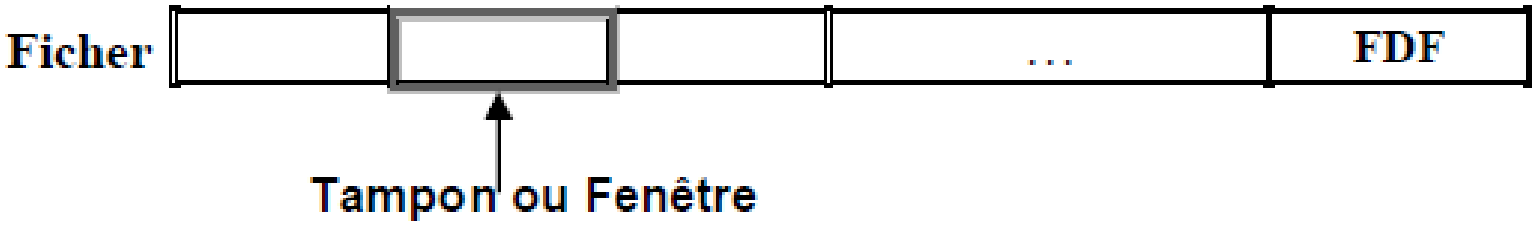
Ce nom est composé de trois parties:

- l'identifiant du support
- le nom du fichier proprement dit
- une extension (ou suffixe) qui précise le genre du fichier (donnée, texte, programme, etc.)

Exemple: **« A:nombres.DAT »** désigne un fichier de données stocké sur la disquette et qui s'appelle nombres.

Eléments attachés à un fichier

- On appelle **tampon ou buffer** d'un fichier, une zone de la mémoire principale pouvant contenir un enregistrement du fichier.
- Un fichier possède toujours un enregistrement supplémentaire à la fin appelé marque de fin de fichier (FDF) permettant de le borner



Eléments attachés à un fichier

Chaque fichier est caractérisé par :

- un **mode d'organisation** : séquentielle, séquentielle indexée, relative ou sélective.
- un **mode d'accès** : séquentiel ou direct

Un fichier à organisation séquentielle (F.O.S) ne permet que l'accès séquentiel : pour atteindre l'article de rang n, il est nécessaire de parcourir les (n-1) articles précédents.

L'accès direct se fait:

- soit en utilisant le rang de l'enregistrement (cas de l'organisation relative) comme dans les tableaux,
- soit en utilisant une clé permettant d'identifier de façon unique chaque enregistrement (cas de l'organisation séquentielle indexée et sélective).

Déclaration d'un fichier à organisation séquentielle

Pour déclarer une variable de type fichier, il faut spécifier :

- le nom du fichier
- le type des éléments du fichier

Exemple

Déclaration d'un fichier de texte

```
Variables
        ftext : Fichier de Caractère
ou
Variables
        ftext : Fichier Texte
```

Déclaration d'un fichier d'enregistrement

```
Types
        Etudiant = Struct
                        Numéro : Entier
                        Nom : Chaîne[30]
                        Prénom : Chaîne[30]
                        Classe : Chaîne[5]
                FinStruct
        Fetud = Fichier de Etudiant
Variables
        Fe : Fetud
```

Manipulation des fichiers à organisation séquentielle

1- Ouverture du fichier :

```
Ouvrir(NomFichier, mode)
```

Un fichier peut être ouvert en mode lecture (L) ou en mode écriture (E).

Après l'ouverture, le pointeur pointe sur le premier enregistrement du fichier.

2- Traitement du fichier :

```
Lire(NomFichier, fenêtre)
```

Cette primitive a pour effet de copier l'enregistrement actuel dans la fenêtre du fichier. Après chaque opération de lecture, le pointeur passe à l'enregistrement suivant.

Manipulation des fichiers à organisation séquentielle

Ecrire(NomFichier, fenêtre)

Cette primitive a pour effet de copier le contenu de la fenêtre sur le fichier en mémoire secondaire. Dans un fichier à organisation séquentielle, l'ajout d'un nouveau article se fait toujours en fin de fichier.

3- Fermeture du fichier :

Fermer(NomFichier)

La fonction booléenne **FDF(NomFichier)** permet de tester si la fin du fichier est atteinte. Sa valeur est déterminée par le dernier ordre de lecture exécuté.

Fichiers de type texte

Les fichiers de texte sont des fichiers séquentiels qui contiennent des caractères organisés en lignes. La présentation sous forme de « ligne » est obtenue grâce à la présence des caractères de contrôle :

- retour chariot (noté souvent CR), de code ASCII 13
- saut de ligne (noté souvent LF) de code ASCII 10

Un fichier texte peut être déclaré de 2 façons différentes

Var:
Ftext: Fichier de Caractère

Ou

Var :
Ftext:Fichier Texte

Fichiers de type texte

- Un fichier de type texte peut être traité ligne par ligne ou caractère par caractère
- Dans un fichier de type texte, la primitive **Lire_Lig(NomFichier,Fenêtre)** permet de lire une ligne du fichier et la transférer dans la fenêtre.
- De même, la fonction booléenne **FDL(NomFichier)** permet de vérifier si le pointeur a atteint la fin d'une ligne

Fichiers de type texte

Exemple: l'algorithme d'une procédure qui lit et affiche le contenu d'un fichier de type texte.

```
ParcoursFichText(ftext : Fichier texte)
Var
ligne : Chaîne
Début
Ouvrir(ftext,L)
Lire(ftext,ligne)
TantQue Non(FDF(ftext)) Faire
Ecrire(ligne)
Lire(ftext,ligne)
Fait
Fermer(ftext)
Fin
```

Ouverture fichier pour lecture

Lecture d'un caractère et le transférer vers la fenêtre ligne

Test de la fin du fichier

Fermeture fichier après la boucle de lecture

CHAPITRE 3

IDENTIFIER LES PARADIGMES ALGORITHMIQUES

1 - Les procédures et les fonctions

Programmation structurée

La résolution d'un problème complexe peut engendrer des milliers de lignes de code

- algorithme long,
- difficile à écrire,
- difficile à interpréter
- difficile à maintenir.

Solution :

méthodologie de résolution :

Programmation Structurée

Modéliser un problème

Définir un algorithme

Apprendre à formuler un traitement

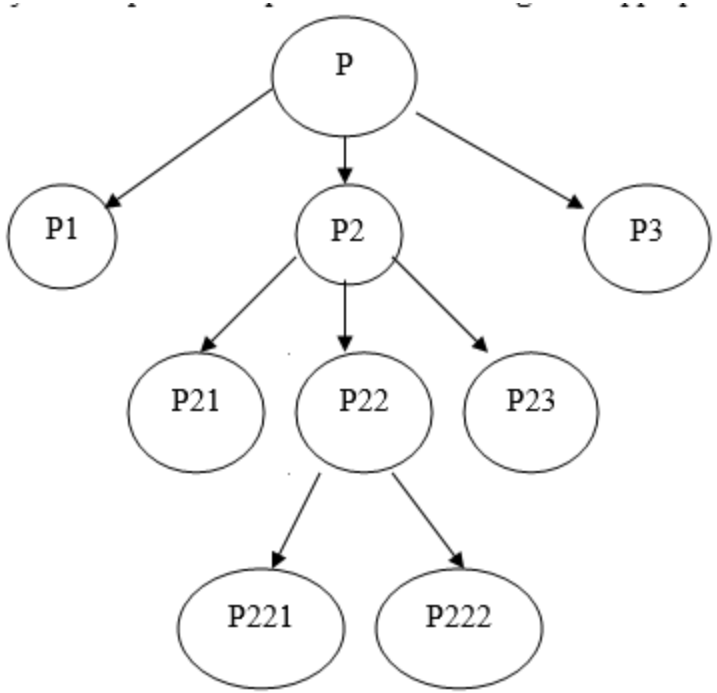
Traduire son algorithme dans un langage de programmation

Programmation structurée

Idée: découpage d'un problème en des sous problèmes moins complexes

Avantages:

- clarté de l'algorithme
- lisibilité de la lecture d'un algorithme
- facilité de maintenance
- réutilisation des sous algorithmes

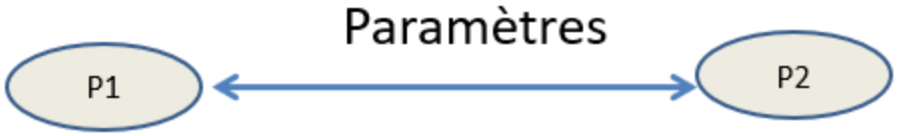


Procédures et fonctions

2 types de sous algorithmes :

- procédure
- fonction

Communication entre sous algorithmes:



paramètres données : les entrées

paramètres résultats : les sorties

paramètres données/résultats : à l'appel des données transformés par la procédure/fonction en résultats

Procédures et fonctions

paramètres formels: objets utilisés pour la description d'un sous algorithme

paramètres effectifs: objets utilisés lors de l'appel d'un sous algorithme

Un paramètre formel est toujours une variable.

Un paramètre effectif peut être :

- une variable
- une constante
- une expression arithmétique
- un appel de fonction

Procédures et fonctions

- Pour tout paramètre formel on fait correspondre un paramètre effectif.

- Le paramètre formel et le paramètre effectif correspondant doivent avoir le même type ou être de types compatibles.
- La correspondance entre paramètres formels et paramètres effectifs se fait selon l'ordre de leurs apparitions dans la définition et dans l'utilisation de la procédure ou la fonction.

Syntaxe Définition d'une procédure

Pour définir une procédure on adoptera la syntaxe suivante :

<Nom_proc>(<liste_par_form>)

Var <declarat_var_locales>

Debut

<Corps_procédure>

Fin

- **<Nom_proc>** : désigne le nom de la procédure.

  <liste_par_form> : la liste des paramètres formels. Un paramètre résultat ou donnée/résultat doit être précédé par le mot clé var.
- **<declarat_var_locales>** : la liste des variables
- **<Corps_procédure>** : la suite des instructions décrivant le traitement à effectuer

Syntaxe Définition d'une procédure

Exemple1 :

La procédure suivante permet de lire N valeurs entières et de calculer la plus petite et la plus grande parmi ces N valeurs. Les entiers saisis doivent être supérieurs à 0 et inférieurs à 100.

- Le nom de cette procédure est **Min_Max**
- Les paramètres formels sont : l'entier N comme paramètre donné, les entiers min et max comme paramètres résultats (précédés par le mot clé var).
- 2 variables locales de type entier : i et x

```
Min_Max( N : entier ; var min, max : entier)
Var   i, x : entier
Début
            min := 100
            max := 0
            pour  i  de  1  à  N  faire
                        Répéter
                            Lire ( x )
                        Jqa  (x > 0) et (x < 100)
                        Si  x < min
                        Alosr  min := x
                        Fsi
                        Si  x > max
                        Alors  max := x
                        Fsi
            Fpour
Fin
```

Syntaxe Définition d'une fonction

définir une fonction on adoptera la syntaxe suivante

<Nom_fonction>(<liste_par_form>) : <Type-fonction>

Var <declarat_var_locales>
Debut

<Corps_fonction>
<Nom_fonc> := <valeur>

Fin

- **<Nom_fonction>, <liste_par_form>, <declarat_var_locales>** définissent les mêmes concepts que pour la procédure
- **<Type-fonction> :** est un type simple.
- En effet, l'un des résultats calculés par la fonction est porté par elle-même. Un type est associé à la fonction caractérisant ainsi ce résultat.
- **<Corps_fonction>** : en plus des instructions décrivant le traitement à effectuer, une instruction d'affectation du résultat que devrait porter la fonction au nom de la fonction elle-même.

Syntaxe Définition d'une fonction

Nous reprenons le problème de l'exemple 1 décrit sous la forme d'une fonction :

```
Min_Max( N : entier ; var min: entier) : entier
Var   i, x, max : entier
Début
            min := 100
            max := 0
            pour  i  de  1  à  N  faire
                        Répéter
                            Lire ( x )
                        Jusqu'à  (x > 0) et (x < 100)
                        Si  x < min
                        Alors  min := x
                        Fsi
                        Si  x > max
                        Alors  max := x
                        Fsi
            Fpour
            Min_Max := max
Fin
```

Modéliser un problème | Définir un algorithme | Apprendre à formuler un traitement | Traduire son algorithme dans un langage de programmation

Syntaxe Utilisation

Lors de l'appel d'un sous algorithme (procédure ou fonction) à partir d'un algorithme appelant, on utilisera le nom de la procédure ou la fonction suivi par la liste de ses paramètres effectifs

<Nom>(<liste_par_effectif>)

- **<Nom> :** est le nom de la procédure ou la fonction
- **<liste_par_effectif> :** une suite d'objets désignant les paramètres effectifs séparés par des virgules (',').

- Les paramètres effectifs et les paramètres formels doivent être compatible en nombre et en type.
- La correspondance entre les 2 types de paramètres se fait selon l'ordre d'apparition.

Syntaxe Utilisation

Exemple: On souhaite écrire un algorithme qui lit un entier N supérieur à 3, puis saisit N valeurs entières et affiche la plus petite et la plus grande parmi ces N valeurs. Les entiers saisis doivent être supérieurs à 0 et inférieurs à 100.

```
Affiche_min_max
Var   N, min, max : entier
Début
        lire_nombre( N )

        max := Min_Max(N, min)

        écrire( ''la plus grande valeur est :'', max)
        écrire( ''la plus petite valeur est :'', min)

Fin
```

Appel de procédure

```
lire_nombre( ) : entier
Var   N : entier
Début
        Répéter
                Lire ( N )
        Jusqu'à  ( N > 3)
        lire_nombre  := N
Fin
```

Appel de la fonction

```
Min_Max( N : entier ; var min: entier) : entier
Var   i, x, max : entier
Début
        min := 100
        max := 0
        pour i de 1 à N faire
                Répéter
                    Lire ( x )
                Jusqu'à (x > 0) et (x < 100)
                Si  x < min
                Alors  min := x
                Fsi
                Si  x > max
                Alors  max := x
                Fsi
        Fpour
        Min_Max := max
Fin
```

Passage de paramètre

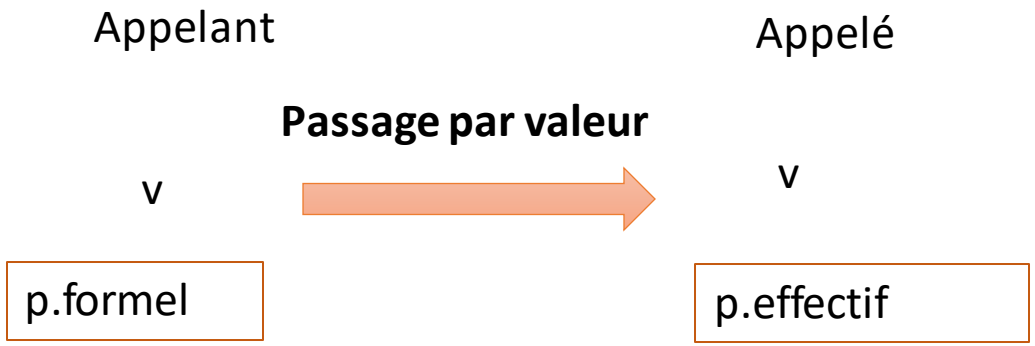
Passage par valeur : valeur du paramètre effectif transmise au paramètre formel.

→ **paramètre donnée**

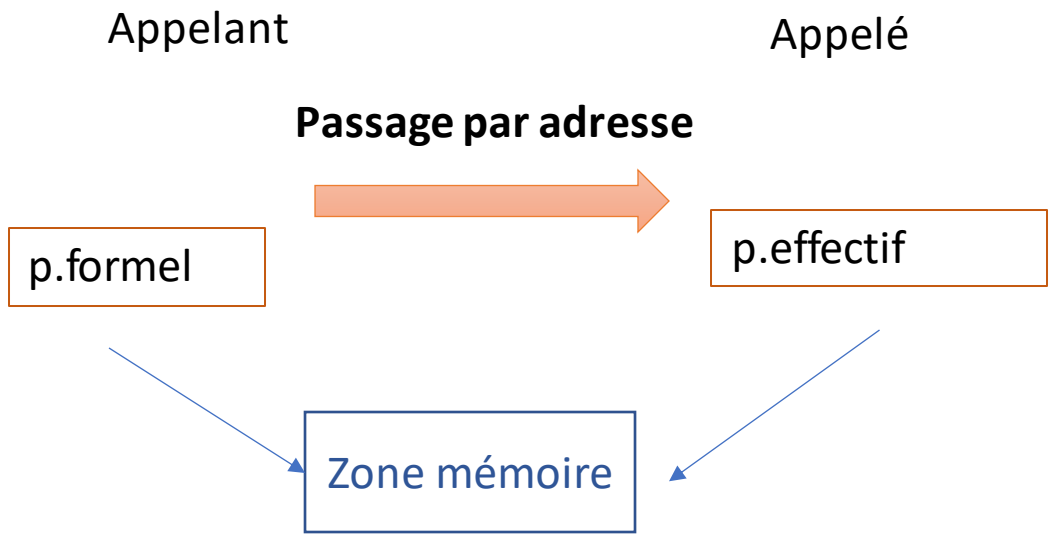
Passage par adresse : l'adresse du paramètre effectif transmise au paramètre formel

→ **paramètre résultat ou donnée/résultat**

Passage de paramètre



- Une copie est transmise
- Toute modification sur le paramètre formel n'altère pas le paramètre effectif



- Une adresse est transmise
- Toute modification sur le paramètre formel altère pas le paramètre effectif

Passage de paramètre

Exemple

Passage de paramètres par valeur

```
ajoute_un(a : entier)
Debut
    a := a+1
Fin
```

Appel :

```
Programme Principal
var  x : entier
Debut
    x := 9
    ajoute_un(x)
    ecrire(x)
Fin
```

Valeur affichée 9

Passage de paramètres par adresse

```
inc(var x : entier)
Debut
    x := x+1
Fin
```

Appel :

```
Programme Principal
var  y : entier
Debut
    y := 100
    inc(y)
    ecrire(y)
Fin
```

Valeur affichée 101

CHAPITRE 3

IDENTIFIER LES PARADIGMES ALGORITHMIQUES

2 – La récursivité

Définition

- Une construction est récursive si elle se définit à **partir d'elle-même.**
- On appelle récursive toute fonction ou procédure qui **s'appelle elle même.**

Exemple:

0 ! = 1

n! = n * [(n – 1) * (n – 2) · · · * 2 * 1] pour n > 0

(n-1) !

En effet,

(n-1) ! = (n – 1) * · · · * 2 * 1

D'où, on peut exprimer n ! avec : **n! = n*(n-1) !**

```
factorielle( n : entier ) : entier
Debut
si  n = 0  alors  factorielle := 1
           sinon factorielle :=  n * factorielle ( n-1 )
fsi
Fin
```

Version récursive de la procédure factorielle

Généralités

Pourquoi étudier la récursivité?

1. La récursivité est un **moyen simple** pour résoudre un problème
2. La récursivité s'apprête à la preuve formelle de correction d'un algorithme
3. La récursivité représente le mécanisme de base pour de nombreux langages de programmation

Quand Utilise_t_on la récursivité?

- la récursivité quand le problème à résoudre se décompose en 2 ou plusieurs sous problèmes de même nature que lui mais de complexité moindre.
- La récursivité s'avérer inefficace si elle est mal utilisée. Il est déconseillé d'utiliser la récursivité quand il existe une définition itérative évidente pour le problème.

Analyse récursive

Pour écrire un algorithme récursif permettant de résoudre un problème P sur des données D, nous suivrons l'approche suivante :

1. **décomposer** le problème P en un ou plusieurs sous problèmes tous de même nature que P mais de complexité moindre
2. **résoudre chaque sous problème** de manière indépendante en suivant la même démarche
3. **déterminer un ou plusieurs cas particuliers** pour lesquels la solution est évidente et tel qu'on ne peut pas appliquer la règle de décomposition
4. **définir une règle de combinaison** qui détermine le résultat du problème P à partir des résultats des sous problèmes

Analyse récursive

Selon cette démarche, le principe d'une analyse récursive d'un problème est basé sur 3 critères:

- une formule de récurrence utilisée lors de la décomposition
- une condition d'arrêt pour la terminaison de la décomposition
- une règle de combinaison pour générer le résultat du problème initial à partir des résultats intermédiaires (résultats des sous problèmes).

<u>Dans l'exemple de la factorielle,</u>

- la formule de récurrence est : **n! = n*(n-1)!**
- Une seule condition d'arrêt qui correspond au cas particulier **n=0**
- La règle de combinaison est de **multiplier la donnée n par le résultat du sous problème**.

Différents types de récursivité

Il existe plusieurs types de récursivité:

1. **récursivité terminale**, la fonction se termine avec l'unique appel récursif.
2. **récursivité multiple** , si l'un des cas traité se résout avec plusieurs appels récursifs.
3. **récursivité croisée ou mutuelle** , deux algorithmes sont mutuellement récursifs si l'un fait appel à l'autre et vice-versa.
4. **récursivité imbriquée**, si la fonction contient comme paramètre un appel à elle-même