

# **Cours d'ASD**

---

**1ère année LISI**

**2022/2023, Semestre 1**

# Algorithmique

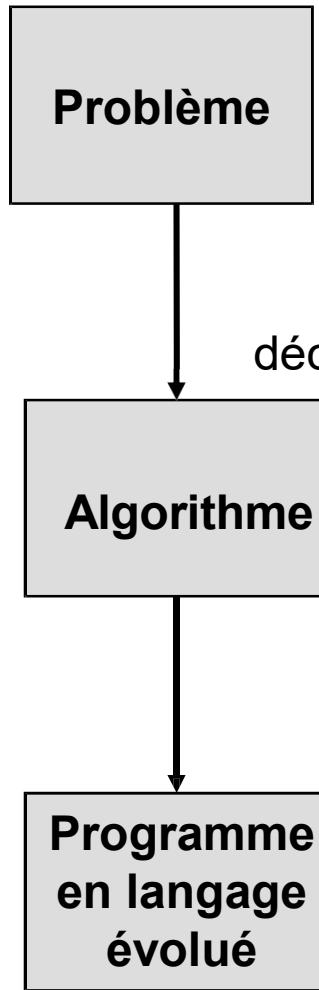
- Le terme **algorithme** vient du nom du mathématicien arabe **Al-Khawarizmi** (820 après J.C.)
- Un algorithme est une description complète et détaillée des actions à effectuer et de leur séquencement pour arriver à un résultat donné
  - Intérêt: séparation analyse/codage (pas de préoccupation de syntaxe)
  - Qualités: **exact** (fournit le résultat souhaité), **efficace** (temps d'exécution, mémoire occupée), **clair** (compréhensible), **général** (traite le plus grand nombre de cas possibles), ...
- **L'algorithmique** désigne aussi la discipline qui étudie les algorithmes et leurs applications en Informatique
- Une bonne connaissance de l'algorithmique permet d'écrire des algorithmes exacts et efficaces

# Représentation d'un algorithme

Historiquement, deux façons pour représenter un algorithme:

- **L'Organigramme:** représentation graphique avec des symboles (carrés, losanges, etc.)
  - offre une vue d'ensemble de l'algorithme
  - représentation quasiment abandonnée aujourd'hui
- **Le pseudo-code:** représentation textuelle avec une série de conventions ressemblant à un langage de programmation (sans les problèmes de syntaxe)
  - plus pratique pour écrire un algorithme
  - représentation largement utilisée

# Démarche pour la résolution d'un problème



C'est la définition précise du problème à résoudre.

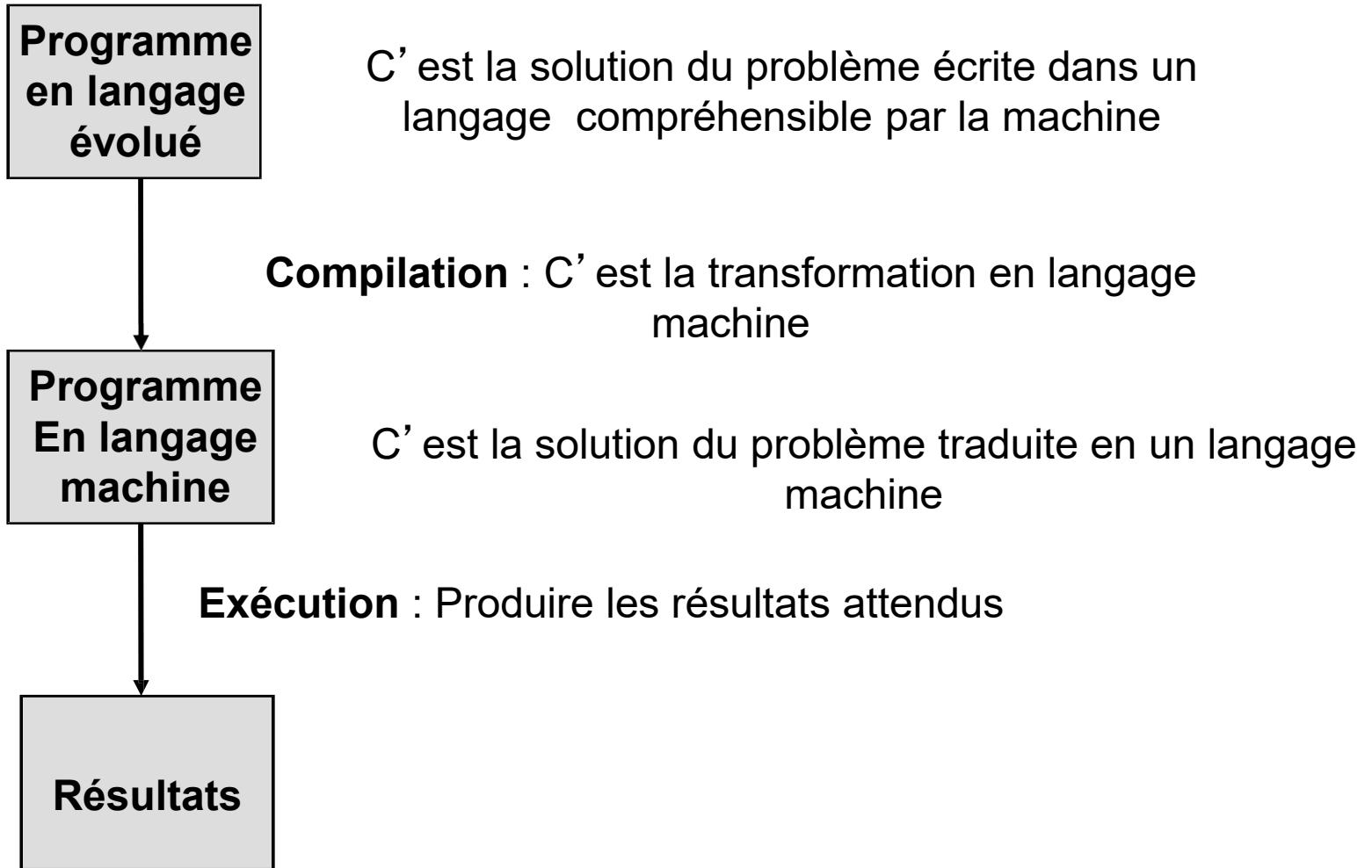
**Analyse** : C'est l'analyse du problème et sa décomposition en sous-problèmes simples et distinctes.

C'est la solution du problème écrite sous une forme textuelle Compréhensible par un être humain.

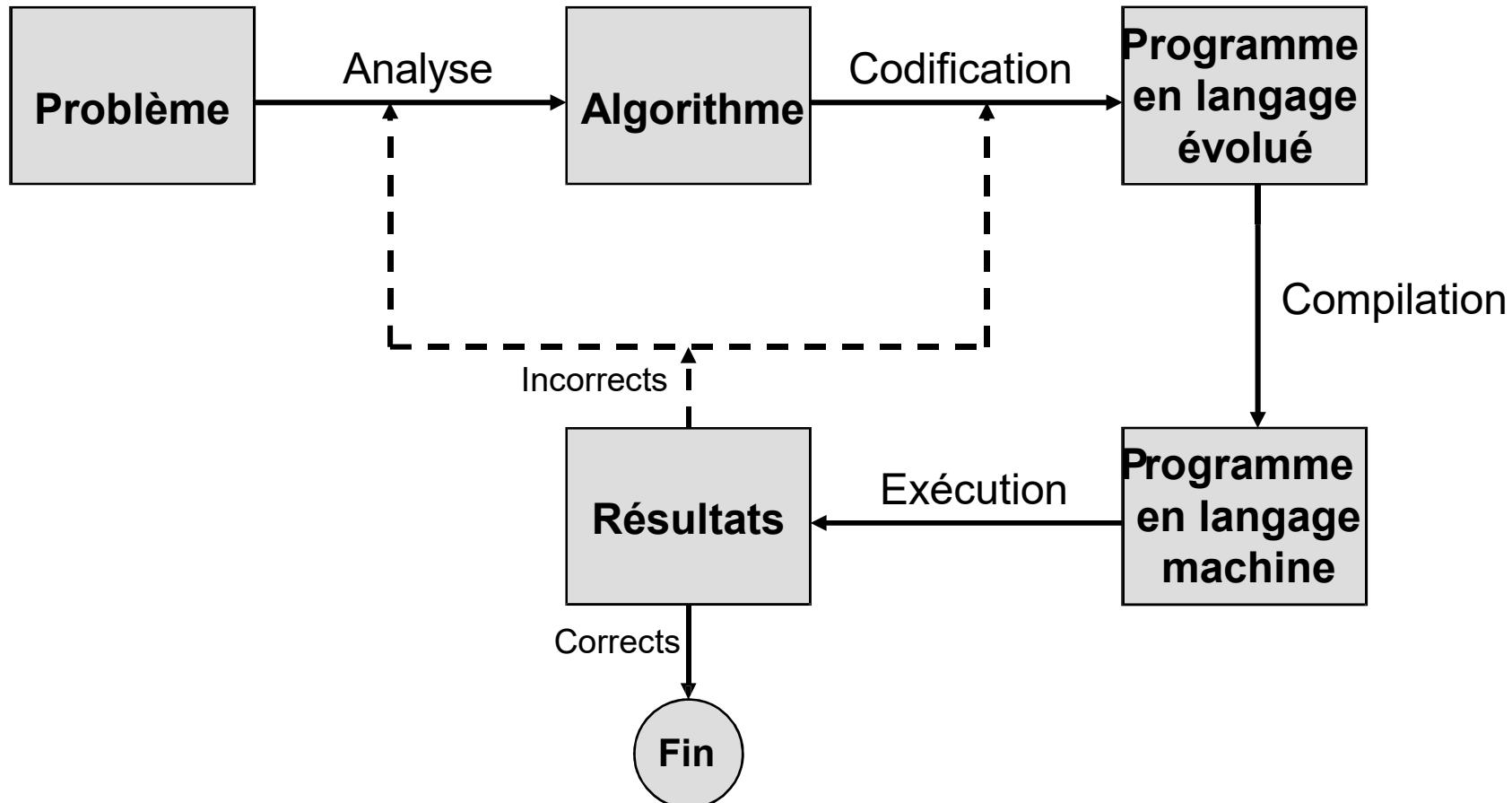
**Codification** : C'est la traduction de l'algorithme dans un langage donné compréhensible par la machine.

C'est la solution du problème écrite dans un langage Informatique compréhensible par la machine

# Démarche pour la résolution d'un problème ...



# Démarche pour la résolution d'un problème ...



# ***Algorithmique***

***Notions et instructions de base***

# **Notion de variable**

---

- Dans les langages de programmation une **variable** sert à stocker la valeur d'une donnée
- Une variable désigne en fait un emplacement mémoire dont le contenu peut changer au cours d'un programme (d'où le nom variable)
- Règle : Les variables doivent être **déclarées** avant d'être utilisées, elles doivent être caractérisées par :
  - un nom (**Identificateur**)
  - un **type** (entier, réel, caractère, chaîne de caractères, ...)

# Choix des identificateurs (1)

Le choix des noms de variables est soumis à quelques règles qui varient selon le langage, mais en général:

## **Choix des identificateurs (2)**

Conseil: pour la lisibilité du code choisir des noms significatifs qui décrivent les données manipulées

exemples: **TotalVentes2004, Prix\_TTC, Prix\_HT**

Remarque: en pseudo-code algorithmique, on va respecter les règles citées, même si on est libre dans la syntaxe

# **Types des variables**

Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre, les types offerts par la plupart des langages sont:

- Type numérique (entier ou réel)
  - **Byte** (codé sur 1 octet): de 0 à 255
  - **Entier court** (codé sur 2 octets) : -32 768 à 32 767
  - **Entier long** (codé sur 4 ou 8 octets)
  - **Réel simple précision** (codé sur 4 octets)
  - **Réel double précision** (codé sur 8 octets)
- Type logique ou booléen: deux valeurs VRAI ou FAUX
- Type caractère: lettres majuscules, minuscules, chiffres, symboles, ...  
**exemples:** 'A', 'a', '1', '?', ...
- Type chaîne de caractère: toute suite de caractères,  
**exemples:** " Nom, Prénom", "code postale: 1000", ...

# Déclaration des variables

- Rappel: toute variable utilisée dans un programme doit avoir fait l'objet d'une déclaration préalable
- En pseudo-code, on va adopter la forme suivante pour la déclaration de variables

**Variables liste d'identificateurs : type**

- Exemple:

**Variables i, j,k : entier**

**x, y : réel**

**OK: booléen**

**ch1, ch2 : chaîne de caractères**

- Remarque: pour le type numérique on va se limiter aux entiers et réels sans considérer les sous types

# **Notion de constante**

---

- Contrairement aux variables, les constantes sont des données dont la valeur reste fixe durant l'exécution du programme.
- La déclaration des constantes peut se faire avant ou après les variables
- En pseudo-code, on va adopter la forme suivante pour la déclaration de constantes

**Constantes      nom\_de\_la\_constante = valeur**

- Exemples:

**Constantes** Pi = 3,14159

ch = 'calcul'

C = 'R'

test = TRUE

# Structure d'un algorithme

**Algorithme Nom\_algorithme**

Déclarations de variables

Déclarations de constantes

**Début**

Instructions

**Fin**

**Nom de l'Algorithme** : c'est l'identification ou le nom.

**Déclarations** : description de tous les objets utilisés dans l'algorithme, définition de variables et de constantes.

**Instructions**: séquence d'actions à exécuter sur l'environnement afin de résoudre le problème. Cette partie est également appelée le corps de l'algorithme.

# Instruction d'affectation

- **l'affectation** consiste à attribuer une valeur à une variable  
(ça consiste en fait à remplir où à modifier le contenu d'une zone mémoire)
- En pseudo-code, l'affectation se note avec le signe  $\leftarrow$   
**Var $\leftarrow$  e: attribue la valeur de e à la variable Var**
  - e peut être une valeur, une autre variable ou une expression
  - Var et e doivent être de même type ou de types compatibles
  - l'affectation ne modifie que ce qui est à gauche de la flèche
- **Ex valides:**     $i \leftarrow 1$                            $j \leftarrow i$                            $k \leftarrow i+j$   
                               $x \leftarrow 10.3$                        $OK \leftarrow FAUX$                        $ch1 \leftarrow "LISI"$   
                               $ch2 \leftarrow ch1$                        $x \leftarrow 4$                                $x \leftarrow j$
- **non valides:**     $i \leftarrow 10.3$                        $OK \leftarrow "LISI"$                        $j \leftarrow x$

(voir la déclaration des variables dans le transparent précédent)

# Quelques remarques

- Beaucoup de langages de programmation (C/C++, Java, ...) utilisent le signe égal  $=$  pour l'affectation  $\leftarrow$ . Attention aux confusions:
  - l'affectation n'est pas commutative :  $A=B$  est différente de  $B=A$
  - l'affectation est différente d'une équation mathématique :
    - $A=A+1$  a un sens en langages de programmation
    - $A+1=2$  n'est pas possible en langages de programmation et n'est pas équivalente à  $A=1$
- Certains langages donnent des valeurs par défaut aux variables déclarées. Pour éviter tout problème il est préférable **d'initialiser les variables** déclarées

# **Exercices simples sur l'affectation (1)**

Donnez les valeurs des variables A, B et C après exécution des instructions suivantes ?

**Variables A, B, C: Entier**

**Début**

A  $\leftarrow$  3

B  $\leftarrow$  7

A  $\leftarrow$  B

B  $\leftarrow$  A+5

C  $\leftarrow$  A + B

C  $\leftarrow$  B - A

**Fin**

## **Exercices simples sur l'affectation (2)**

Donnez les valeurs des variables A et B après exécution des instructions ?

<b>Variables</b>	A,	B	:	<b>Entier</b>
A		←		1
B		←		2
A		←		B
B		←		A
		<b>Fin</b>		

Les deux dernières instructions permettent-elles d'échanger les valeurs de A et B ?

## **Exercices simples sur l'affectation (3)**

---

Ecrire un algorithme permettant d'échanger les valeurs de deux variables A et B

# Expressions et opérateurs

- Une **expression** peut être une valeur, une variable ou une opération constituée de variables reliées par des **opérateurs**  
**exemples:** 1, b, a\*2, a+ 3\*b-c, ...
- L'évaluation de l'expression fournit une valeur unique qui est le résultat de l'opération
- Les **opérateurs** dépendent du type de l'opération, ils peuvent être :
  - des opérateurs arithmétiques: +, -, \*, /, % (modulo), ^ (puissance)
  - des opérateurs logiques: NON, OU, ET
  - des opérateurs relationnels: =, ≠, <, >, <=, >=
  - des opérateurs sur les chaînes: & (concaténation)
- Une expression est évaluée de gauche à droite mais en tenant compte de **priorités**

# Priorité des opérateurs

- Pour les opérateurs arithmétiques donnés ci-dessus, l'ordre de priorité est le suivant (du plus prioritaire au moins prioritaire) :

- $^$  : (élévation à la puissance)
- $*$ ,  $/$  (multiplication, division)
- $\%$  (modulo)
- $+$ ,  $-$  (addition, soustraction)

**exemple:**  $2 + 3 * 7$  vaut 23

- En cas de besoin (ou de doute), on utilise les parenthèses pour indiquer les opérations à effectuer en priorité

**exemple:**  $(2 + 3) * 7$  vaut 35

# Les instructions d'entrées-sorties: lecture et écriture (1)

- Les instructions de lecture et d'écriture permettent à la machine de communiquer avec l'utilisateur
- La **lecture** permet d'entrer des données à partir du clavier
  - En pseudo-code, on note: **Lire (var)**  
la machine met la valeur entrée au clavier dans la zone mémoire nommée var
  - Remarque: Le programme s'arrête lorsqu'il rencontre une instruction Lire et ne se poursuit qu'après la frappe d'une valeur au clavier et de la touche Entrée

# Les instructions d'entrées-sorties: lecture et écriture (2)

- L'**écriture** permet d'afficher des résultats à l'écran (ou de les écrire dans un fichier)
  - En pseudo-code, on note: **Ecrire (var)**  
la machine affiche le contenu de la zone mémoire var
  - Conseil: Avant de lire une variable, il est fortement conseillé d'écrire des messages à l'écran, afin de prévenir l'utilisateur de ce qu'il doit frapper

# **Exemple (lecture et écriture)**

Ecrire un algorithme qui demande un nombre entier à l'utilisateur, puis qui calcule et affiche le double de ce nombre

**Algorithme Calcul\_double**

**Variables A, B : entier**

**Début**

```
Ecrire("entrer le nombre ")  
Lire(A)  
B ← 2*A  
Ecrire("le double de ", A, "est :", B)
```

**Fin**

# *Algorithmique*

***Structures Conditionnelles***

## Tests: instructions conditionnelles (1)

- Les instructions conditionnelles servent à n'exécuter une instruction ou une séquence d'instructions que si une condition est vérifiée
  - On utilisera la forme suivante: **Si** condition **alors**  
instruction ou suite d'instructions1  
**Sinon**  
instruction ou suite d'instructions2  
**Finsi**
  - la condition ne peut être que vraie ou fausse
  - si la condition est vraie, se sont les instructions1 qui seront exécutées
  - si la condition est fausse, se sont les instructions2 qui seront exécutées
  - la condition peut être une condition simple ou une condition composée de plusieurs conditions

## Tests: instructions conditionnelles (2)

- La partie Sinon n'est pas obligatoire, quand elle n'existe pas et que la condition est fausse, aucun traitement n'est réalisé
  - On utilisera dans ce cas la forme simplifiée suivante:

**Si** condition **alors**

instruction ou suite d'instructions1

**Finsi**

# **Exemple (Si...Alors...Sinon)**

**Algorithme AffichageValeurAbsolue (version1)**

**Variable x : réel**

**Début**

**Ecrire (" Entrez un réel : ")**

**Lire (x)**

**Si (x < 0) alors**

**Ecrire ("la valeur absolue de ", x, "est:",-x)**

**Sinon**

**Ecrire ("la valeur absolue de ", x, "est:",x)**

**Finsi**

**Fin**

# **Exemple (Si...Alors)**

**Algorithme AffichageValeurAbsolue (version2)**

**Variable** x,y : réel

**Début**

**Ecrire** (" Entrez un réel :")

**Lire** (x)

**y**  $\leftarrow$  x

**Si** (x < 0) **alors**

**y**  $\leftarrow$  -x

**Finsi**

**Ecrire** ("la valeur absolue de ", x, "est:",y)

**Fin**

# **Exercice (tests)**

Ecrire un algorithme qui demande un nombre entier à l'utilisateur, puis qui teste et affiche s'il est divisible par 3

**Algorithme Divsible\_par3**

**Variable n : entier**

**Début**

**Ecrire (" Entrez un entier : ")**

**Lire (n)**

**Si (n%3=0) alors**

**Ecrire (n," est divisible par 3")**

**Sinon**

**Ecrire (n," n'est pas divisible par 3")**

**Finsi**

**Fin**

# Conditions composées

- Une condition composée est une condition formée de plusieurs conditions simples reliées par des opérateurs logiques:  
ET, OU, OU exclusif (XOR) et NON
- Exemples :
  - x compris entre 2 et 6 :  $(x > 2)$  ET  $(x < 6)$
  - n divisible par 3 ou par 2 :  $(n \% 3 = 0)$  OU  $(n \% 2 = 0)$
  - deux valeurs et deux seulement sont identiques parmi a, b et c :  
 $(a=b)$  XOR  $(a=c)$  XOR  $(b=c)$
- L'évaluation d'une condition composée se fait selon des règles présentées généralement dans ce qu'on appelle tables de vérité

# Tables de vérité

C1	C2	C1 ET C2
VRAI	VRAI	<b>VRAI</b>
VRAI	FAUX	<b>FAUX</b>
FAUX	VRAI	<b>FAUX</b>
FAUX	FAUX	<b>FAUX</b>

C1	C2	C1 OU C2
VRAI	VRAI	<b>VRAI</b>
VRAI	FAUX	<b>VRAI</b>
FAUX	VRAI	<b>VRAI</b>
FAUX	FAUX	<b>FAUX</b>

C1	C2	C1 XOR C2
VRAI	VRAI	<b>FAUX</b>
VRAI	FAUX	<b>VRAI</b>
FAUX	VRAI	<b>VRAI</b>
FAUX	FAUX	<b>FAUX</b>

C1	NON C1
VRAI	<b>FAUX</b>
FAUX	<b>VRAI</b>

# Tests imbriqués

- Les tests peuvent avoir un degré quelconque d'imbrications

**Si condition1 alors**

**Si condition2 alors**

instructionsA

**Sinon**

instructionsB

**Finsi**

**Sinon**

**Si condition3 alors**

instructionsC

**Finsi**

**Finsi**

# Tests imbriqués: exemple (version 1)

Variable n : entier

**Début**

Ecrire ("entrez un nombre : ")

Lire (n)

**Si (n < 0) alors**

Ecrire ("Ce nombre est négatif")

**Sinon**

**Si (n = 0) alors**

Ecrire ("Ce nombre est nul")

**Sinon**

Ecrire ("Ce nombre est positif")

**Finsi**

**Finsi**

**Fin**

# Tests imbriqués: exemple (version 2)

Variable n : entier

**Début**

Ecrire ("entrez un nombre : ")

Lire (n)

**Si** (n < 0) **alors** Ecrire ("Ce nombre est négatif")

**Finsi**

**Si** (n = 0) **alors** Ecrire ("Ce nombre est nul")

**Finsi**

**Si** (n > 0) **alors** Ecrire ("Ce nombre est positif")

**Finsi**

**Fin**

**Remarque :** dans la version 2 on fait trois tests systématiquement alors que dans la version 1, si le nombre est négatif on ne fait qu'un seul test

**Conseil :** utiliser les tests imbriqués pour limiter le nombre de tests et placer d'abord les conditions les plus probables

# **Structure à choix multiple (1)**

## Syntaxe générale:

**SELON Sélecteur FAIRE**

**Liste\_valeurs1 : Traitement 1**

**Liste\_valeurs2 : Traitement 2**

...

**Liste\_valeursN : Traitement N**

**Autre : Traitement**

**FIN SELON**

**Sélecteur** : C'est un identificateur de variable ou une expression de type scalaire.

**Liste\_valeurs1..Liste\_valeursN** : Ce sont des valeurs servant aux tests. Elles peuvent être données sous forme de constantes et/ou d'intervalles constantes de type compatible avec le sélecteur.

## **Structure à choix multiple (2)**

### **Application:**

Ecrire un algorithme qui permet de lire un numéro compris entre 1 et 12 et d'afficher le nom du mois correspondant. Si le numéro entré est en dehors de cet intervalle, un message d'erreur doit être affiché.

# Structure à choix multiple (3)

## Solution

**Algorithme** mois

**Variables**

n:Entier

**Début**

Ecrire("Entrer le numéro du mois:")

Lire(n)

**Selon** n **Faire**

1: Ecrire("janvier")

2: Ecrire("février")

...

**Autre:** Ecrire(numéro de mois erroné...)

**FinSelon**

**Fin**

# **Structure à choix multiple (4)**

**Application 2** : algorithme permettant à partir d'un menu affiché à l'écran, d'effectuer la somme ou le produit ou la moyenne de trois nombres.

## **Solution**

### **Algorithme Menu**

**Var**

nb1, nb2, nb3 : Réel

choix : Entier

### **Début**

Ecrire ("Donner trois nombres")

Lire (nb1, nb2, nb3)

-- Affichage du menu et saisie du choix

Ecrire (" 1- pour la multiplication")

Ecrire (" 2- pour la somme")

Ecrire (" 3- pour la moyenne")

Ecrire (" Votre choix :")

Lire (choix)

### **Selon (choix) faire**

1 : Ecrire ("le produit des trois nombres est : " , nb1\*nb2\*nb3)

2 : Ecrire ("la somme des trois nombres est : " , nb1+nb2+nb3)

3 : Ecrire ("la moyenne des trois nombres est : " , (nb1+nb2+nb3)/3)

Autre : Ecrire ("saisie de choix incorrecte")

### **Fin Selon**

**Fin**

# ***Algorithmique***

## ***Structures Itératives***

# Instructions itératives: les boucles

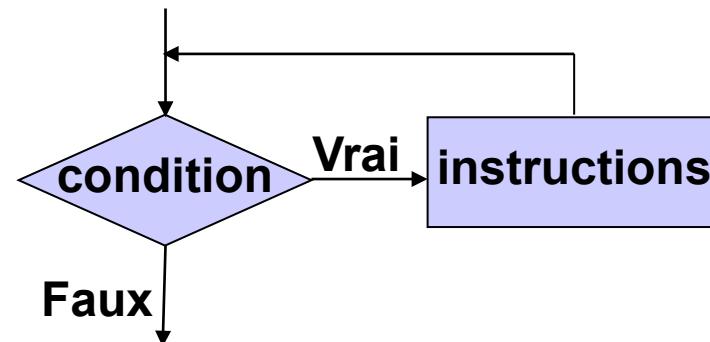
- Les boucles servent à répéter l'exécution d'un groupe d'instructions un certain nombre de fois
- On distingue trois sortes de boucles en langages de programmation :
  - Les **boucles tant que** : on y répète des instructions tant qu'une certaine condition est réalisée
  - Les **boucles jusqu'à** : on y répète des instructions jusqu'à ce qu'une certaine condition soit réalisée
  - Les **boucles pour** ou avec compteur : on y répète des instructions en faisant évoluer un compteur (variable particulière) entre une valeur initiale et une valeur finale

# Les boucles Tant que (1)

**TantQue** (condition) Faire

instructions

**FinTantQue**



- la condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération
- si la condition est vraie, on exécute instructions (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution, ...
- si la condition est fausse, on sort de la boucle et on exécute l'instruction qui est après FinTantQue

## **Les boucles Tant que : remarques (2)**

- Le nombre d'itérations dans une boucle TantQue n'est pas connu au moment d'entrée dans la boucle. Il dépend de l'évolution de la valeur de condition
- Une des instructions du corps de la boucle doit absolument changer la valeur de condition de vrai à faux (après un certain nombre d'itérations), sinon le programme tourne indéfiniment

**⇒Attention aux boucles infinies**

- Exemple de boucle infinie :

i ← 2

TantQue (i > 0) Faire

i ← i+1      (attention aux erreurs de frappe : + au lieu de -)

FinTantQue

## **Boucle Tant que : exemple1 (3)**

Contrôle de saisie d'une lettre majuscule jusqu'à ce que le caractère entré soit valable

Variable C : caractère

**Debut**

Ecrire (" Entrez une lettre majuscule ")

Lire (C)

**TantQue (C < 'A' ou C > 'Z') Faire**

Ecrire ("Saisie erronée. Recommencez")

Lire (C)

**FinTantQue**

Ecrire ("Saisie valable")

**Fin**

# Boucle Tant que : exemple2 (4)

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100

## version 1

Variables som, i : entier

## Debut

i ↑ 0

**som**  $\leftarrow$  0

## TantQue (som <=100) Faire

i ↑ i+1

`som ← som+i`

# FinTantQue

Ecrire (" La valeur cherchée est ", i)

Fin

## **Boucle Tant que : exemple2 (version2) (5)**

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100

version 2: attention à l'ordre des instructions et aux valeurs initiales

Variables som, i : entier

**Début**

    som  $\leftarrow$  0

    i  $\leftarrow$  1

**TantQue** (som  $\leq$  100) **Faire**

        som  $\leftarrow$  som + i

        i  $\leftarrow$  i+1

**FinTantQue**

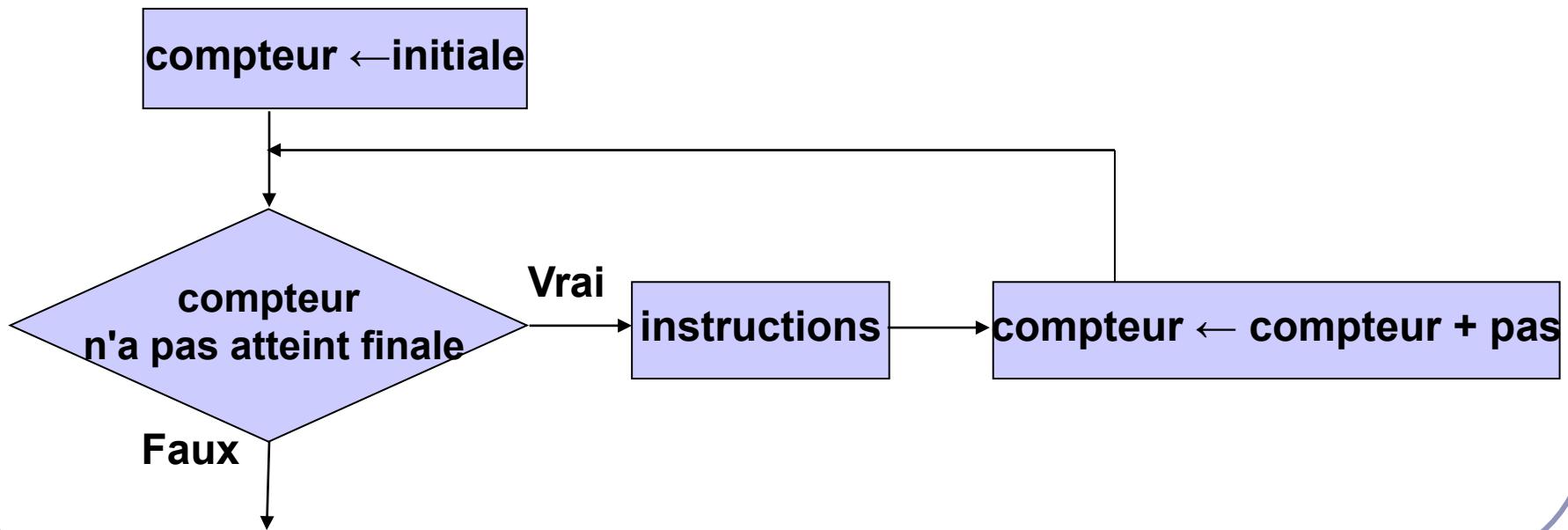
Ecrire (" La valeur cherchée est ", i-1)

**Fin**

# Les boucles Pour (1)

**Pour** compteur **de** initiale **à** finale par **pas** <valeur du pas> **Faire**  
instructions

**FinPour**



## Les boucles Pour (2)

- Remarque : le nombre d'itérations dans une boucle Pour est connu avant le début de la boucle
- **Compteur** est une variable de type entier (ou caractère). Elle doit être déclarée
- **Pas** est un entier qui peut être positif ou négatif. **Pas** peut ne pas être mentionné, car par défaut sa valeur est égale à 1. Dans ce cas, le nombre d'itérations est égal à finale - initiale+ 1
- **Initiale et finale** peuvent être des valeurs, des variables définies avant le début de la boucle ou des expressions de même type que compteur

## **Déroulement des boucles Pour (3)**

- 1) La valeur initiale est affectée à la variable compteur
- 2) On compare la valeur du compteur et la valeur de finale :
  - a) Si la valeur du compteur est  $>$  à la valeur finale dans le cas d'un pas positif (ou si compteur est  $<$  à finale pour un pas négatif), on sort de la boucle et on continue avec l'instruction qui suit FinPour
  - b) Si compteur est  $\leq$  à finale dans le cas d'un pas positif (ou si compteur est  $\geq$  à finale pour un pas négatif), instructions seront exécutées
    - i. Ensuite, la valeur de compteur est incrémentée de la valeur du pas si pas est positif (ou décrémenté si pas est négatif)
    - ii. On recommence l'étape 2 : La comparaison entre compteur et finale est de nouveau effectuée, et ainsi de suite ...

# Boucle Pour : exemple1 (4)

Calcul de  $x$  à la puissance  $n$  où  $x$  est un réel non nul et  $n$  un entier positif ou nul

Variables  $x$ , puiss : réel  
 $n$ ,  $i$  : entier

**Debut**

Ecrire (" Entrez la valeur de  $x$  ")

Lire ( $x$ )

Ecrire (" Entrez la valeur de  $n$  ")

Lire ( $n$ )

puiss  $\leftarrow 1$

**Pour  $i$  de 1 à  $n$  Faire**

puiss  $\leftarrow$  puiss \*  $x$

**FinPour**

Ecrire ( $x$ , " à la puissance ",  $n$ , " est égal à ", puiss)

**Fin**

# Boucle Pour : exemple1 (version 2) (5)

Calcul de  $x$  à la puissance  $n$  où  $x$  est un réel non nul et  $n$  un entier positif ou nul (**version 2 avec un pas négatif**)

Variables  $x$ , puiss : réel

$n$ ,  $i$  : entier

**Debut**

Ecrire (" Entrez respectivement les valeurs de  $x$  et  $n$  ")

Lire ( $x$ ,  $n$ )

puiss  $\leftarrow$  1

**Pour  $i$  de  $n$  à 1 par pas -1 Faire**

puiss  $\leftarrow$  puiss \*  $x$

**FinPour**

Ecrire ( $x$ , " à la puissance ",  $n$ , " est égal à ", puiss)

**Fin**

## Boucle Pour : remarque (6)

- Il faut éviter de modifier la valeur du compteur (et de finale) à l'intérieur de la boucle. En effet, une telle action :
  - perturbe le nombre d'itérations prévu par la boucle Pour
  - rend difficile la lecture de l'algorithme
  - présente le risque d'aboutir à une boucle infinie

Exemple : **Pour i de 1 à 5 Faire**

i                    ←                    i                    -1

écrire(" i = ", i)

**FinPour**

# Lien entre Pour et TantQue (1)

La boucle Pour est un cas particulier de Tant Que (cas où le nombre d'itérations est connu et fixé) . Tout ce qu'on peut écrire avec Pour peut être remplacé avec TantQue (la réciproque est fausse)

**Pour** compteur **de** initiale **à** finale par **pas** valeur du pas **Faire**  
instructions

**FinPour**

peut être remplacé par :  
(cas d'un pas positif)

compteur  $\leftarrow$  initiale  
**TantQue** compteur  $\leq$  finale **Faire**  
instructions  
compteur  $\leftarrow$  compteur+pas

**FinTantQue**

## Lien entre Pour et TantQue: exemple (2)

Calcul de  $x$  à la puissance  $n$  où  $x$  est un réel non nul et  $n$  un entier positif ou nul (**version avec TantQue**)

Variables  $x$ , puiss : réel  
 $n$ ,  $i$  : entier

**Debut**

Ecrire (" Entrez la valeur de  $x$  ")

Lire ( $x$ )

Ecrire (" Entrez la valeur de  $n$  ")

Lire ( $n$ )

puiss  $\leftarrow 1$

$i \leftarrow 1$

**TantQue** ( $i \leq n$ ) **Faire**

    puiss  $\leftarrow$  puiss \*  $x$

$i \leftarrow i + 1$

**FinTantQue**

Ecrire ( $x$ , " à la puissance ",  $n$ , " est égal à ", puiss)

**Fin**

# Boucles imbriquées

- Les instructions d'une boucle peuvent être des instructions itératives. Dans ce cas, on aboutit à des **boucles imbriquées**

- Exemple:**

Pour i de 1 à 5 Faire

    Pour j de 1 à i Faire

        écrire("O")

    FinPour

    écrire("X")

FinPour

**Exécution**

OX

OOX

OOOX

OOOOX

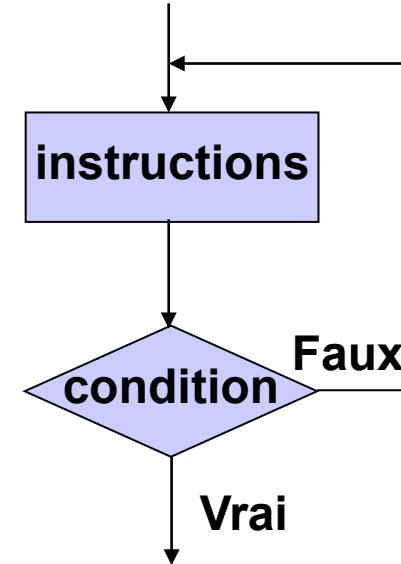
OOOOOX

# Les boucles Répéter ... jusqu'à ... (1)

Répéter

instructions

Jusqu'à condition



- Condition est évaluée après chaque itération
- les instructions entre *Répéter* et *jusqu'à* sont exécutées au moins une fois et leur exécution est répétée jusqu'à ce que condition soit vraie (tant qu'elle est fausse)

## Boucle Répéter jusqu'à : exemple (2)

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100 (**version avec répéter jusqu'à**)

Variables som, i : entier

**Debut**

    som  $\leftarrow$  0

    i  $\leftarrow$  0

**Répéter**

    i  $\leftarrow$  i+1

    som  $\leftarrow$  som+i

**Jusqu'à ( som > 100)**

    Ecrire (" La valeur cherchée est = ", i)

**Fin**

# Choix d'un type de boucle

- Si on peut déterminer le nombre d'itérations avant l'exécution de la boucle, il est plus naturel d'utiliser *la boucle Pour*
- S'il n'est pas possible de connaître le nombre d'itérations avant l'exécution de la boucle, on fera appel à l'une des *boucles TantQue ou répéter jusqu'à*
- Pour le choix entre *TantQue* et *jusqu'à* :
  - Si on doit tester la condition de contrôle avant de commencer les instructions de la boucle, on utilisera *TantQue*
  - Si la valeur de la condition de contrôle dépend d'une première exécution des instructions de la boucle, on utilisera *répéter jusqu'à*

# ***ALGORITHMIQUE***

***Les tableaux***

# Exemple introductif

- Supposons qu'on veut conserver les notes d'une classe de 30 étudiants pour extraire quelques informations. Par exemple : calcul du nombre d'étudiants ayant une note supérieure à 10
- Le seul moyen dont nous disposons actuellement consiste à déclarer 30 variables, par exemple **N1**, ..., **N30**. Après 30 instructions lire, on doit écrire 30 instructions Si pour faire le calcul

**nbre**  $\leftarrow$  0

**Si (N1 >10) alors nbre  $\leftarrow$ nbre+1 FinSi**

....

**Si (N30>10) alors nbre  $\leftarrow$ nbre+1 FinSi**

c'est lourd à écrire

- Heureusement, les langages de programmation offrent la possibilité de rassembler toutes ces variables dans **une seule structure de donnée** appelée **tableau**

# Tableaux

- Un **tableau** est un ensemble d'éléments de même type désignés par un identificateur unique
- Une variable entière nommée **indice** permet d'indiquer la position d'un élément donné au sein du tableau et de déterminer sa valeur
- La **déclaration** d'un tableau s'effectue en précisant le **type** de ses éléments et sa **dimension** (le nombre de ses éléments)
  - En pseudo code :  
Nom : tableau [min..max] de <Type>

Avec:

**Nom** : Nom du tableau

**Min** : indice minimum du tableau

**Max** : indice maximum du tableau **Type** : Type des éléments du tableau

# Tableaux : remarques

- L'accès à un élément du tableau se fait au moyen de l'indice. Par exemple, **notes[i]** donne la valeur de l'élément i du tableau notes
- Il est possible de déclarer un tableau sans préciser au départ sa dimension. Cette précision est faite ultérieurement
  - Par exemple, quand on déclare un tableau comme paramètre d'une procédure, on peut ne préciser sa dimension qu'au moment de l'appel
  - En tous cas, un tableau est inutilisable tant qu'on n'a pas précisé le nombre de ses éléments
- Un grand avantage des tableaux est qu'on peut traiter les données qui y sont stockées de façon simple en utilisant des boucles

# Tableaux : exemple

Pour le calcul du nombre d'étudiants ayant une note supérieure à 10 avec les tableaux, on peut écrire :

Variables            i ,nbre : entier  
                      notes: tableau [1..30] de réels

**Début**

                      nbre  $\leftarrow$  0

**Pour** i de 1 à 30 **Faire**

**Si** (notes[i] >10) alors

                          nbre  $\leftarrow$  nbre+1

**FinSi**

**FinPour**

                      écrire ("le nombre de notes supérieures à 10 est : ", nbre)

**Fin**

# Tableau à deux dimensions

## Déclaration:

- En pseudo code :

Nom : tableau [min1..max1, Min2..Max2] de <Type>

- Avec:

**Nom** : Nom du tableau

**Min1** : indice minimum des lignes

**Max1** : indice maximum des lignes

**Min2** : indice minimum des colonnes

**Max2** : indice maximum des colonnes Type : Type des éléments du tableau.

## Exemple

M : tableau [1..3, 1..4] de réels

M	1	4	
1			
3			

# Tableau à deux dimensions

Accès aux éléments :

Syntaxe :

**Nom [N° \_Ligne, N° \_Colonne]**

Exemples :

//Affecter 22 à l'élément de la ligne 2 et de la colonne 3

M[2,3] □ 22

//Saisie de l'élément de la ligne 4 et de la colonne 1

Lire (M[4,1])

//Affichage de l'élément de la ligne 4 et de la colonne 1

Écrire (M[4,1])

Représentation d'une matrice M

M[1,1]	M[1,2]	M[1,3]	M[1,4]
M[2,1]	M[2,2]	M[2,3]	M[2,4]
M[3,1]	M[3,2]	M[3,3]	M[3,4]

# Tableau à deux dimensions

- Lecture : tout comme le tableau à 1 dimension, il faut faire la lecture case par case

**Pour i = 1 à n Faire**

**Pour j = 1 à m Faire**

**Lire( T[i,j] )**

**Fin Pour**

**Fin Pour**

**Avec:**

n: nombre de lignes

m: nombre de colonnes

# Tableau à 2 dimensions

- Soit T la matrice à 2 lignes et 3 colonnes suivantes:

3	4	1
-1	6	0

- Que vont afficher les instructions suivantes?

Pour i de 1 à 3 Faire

    Pour j de 1 à 2 Faire

        Ecrire (T[j,i])

    Fin Pour

Fin Pour

Pour i de 1 à 2 Faire

    Pour j de 1 à 3 Faire

        Ecrire (T[i,j])

    Fin Pour

Fin Pour

# ***ALGORITHMIQUE***

***Fonctions et procédures***

# Fonctions et procédures

- Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes** ou **modules**
- Les **fonctions** et les **procédures** sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs **intérêts** :
  - permettent de "**factoriser**" **les programmes**, càd de mettre en commun les parties qui se répètent
  - permettent une **structuration** et une **meilleure lisibilité** des programmes
  - **facilitent la maintenance** du code (il suffit de modifier une seule fois)
  - ces procédures et fonctions peuvent éventuellement être **réutilisées** dans d'autres programmes

# Fonctions

- Le **rôle** d'une fonction en programmation est similaire à celui d'une fonction en mathématique : elle **retourne un résultat à partir des valeurs des paramètres**
- Une fonction s'écrit en dehors du programme principal sous la forme :

**Fonction nom\_fonction (paramètres et leurs types) : type\_fonction**

Instructions constituant le corps de la fonction

**retourner ...**

**FinFonction**

- Pour le choix d'un nom de fonction il faut respecter les mêmes règles que celles pour les noms de variables
- type\_fonction est le type du résultat retourné
- L'instruction **retourner** sert à retourner la valeur du résultat

# Fonctions : exemples

- La fonction SommeCarre suivante calcule la somme des carrés de deux réels x et y :

**Fonction** SommeCarre (x : réel, y: réel ) : réel

variable z : réel

$z \leftarrow x^*x+y^*y$

**retourner** (z)

**FinFonction**

- La fonction Pair suivante détermine si un nombre est pair :

**Fonction** Pair (n : entier ) : booléen

**retourner** ( $n\%2=0$ )

**FinFonction**

# Utilisation des fonctions

- L'utilisation d'une fonction se fera par simple écriture de son nom dans le programme principale. Le résultat étant une valeur, devra être affecté ou être utilisé dans une expression, une écriture, ...
- **Exemples : Algorithme exempleAppelFonction**

variables    z : réel, b : booléen

**Début**

b ← Pair(3)

z ← 5 \* SommeCarre(7,2) + 1

écrire("SommeCarre(3,5)= ", SommeCarre(3,5))

**Fin**

- Lors de l'appel Pair(3) le **paramètre formel** n est remplacé par le **paramètre effectif** 3

# Procédures

- Dans certains cas, on peut avoir besoin de répéter une tache dans plusieurs endroits du programme, mais que dans cette tache on ne calcule pas de résultats ou qu'on calcule plusieurs résultats à la fois
- Dans ces cas on ne peut pas utiliser une fonction, on utilise une **procédure**
- Une **procédure** est un sous-programme semblable à une fonction mais qui **ne retourne rien**
- Une procédure s'écrit en dehors du programme principal sous la forme :

**Procédure** nom\_procédure (paramètres et leurs types)

Instructions constituant le corps de la procédure

**FinProcédure**

- Remarque : une procédure peut ne pas avoir de paramètres

# Appel d'une procédure

- L'appel d'une procédure, se fait dans le programme principal ou dans une autre procédure par une instruction indiquant le nom de la procédure :

**Procédure** exemple\_proc (...)

...

**FinProcédure**

**Algorithme exempleAppelProcédure**

**Début**

    exemple\_proc (...)

...

**Fin**

- Remarque : contrairement à l'appel d'une fonction, on ne peut pas affecter la procédure appelée ou l'utiliser dans une expression. L'appel d'une procédure est une instruction autonome

# Paramètres d'une procédure

- Les paramètres servent à échanger des données entre le programme principale (ou la procédure appelante) et la procédure appelée
- Les paramètres placés dans la déclaration d'une procédure sont appelés **paramètres formels**. Ces paramètres peuvent prendre toutes les valeurs possibles mais ils sont abstraits (n'existent pas réellement)
- Les paramètres placés dans l'appel d'une procédure sont appelés **paramètres effectifs**. Ils contiennent les valeurs pour effectuer le traitement
- Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels. L'ordre et le type des paramètres doivent correspondre

# Transmission des paramètres

Il existe deux modes de transmission de paramètres dans les langages de programmation :

- **La transmission par valeur** : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la procédure. Dans ce mode le paramètre effectif ne subit aucune modification
- **La transmission par adresse (ou par référence)** : les adresses des paramètres effectifs sont transmises à la procédure appelante. Dans ce mode, le paramètre effectif subit les mêmes modifications que le paramètre formel lors de l'exécution de la procédure
  - **Remarque** : le paramètre effectif doit être une variable (et non une valeur) lorsqu'il s'agit d'une transmission par adresse
  - En pseudo-code, les paramètres passés par adresse sont précédées du mot clé **Var**

# Transmission des paramètres : exemples

**Procédure** incrementer1 (**x** : entier, **Var y** : entier)

$x \leftarrow x+1$

$y \leftarrow y+1$

**FinProcédure**

**Algorithme Test\_incrementer1**

variables n, m : entier

**Début**

$n \leftarrow 3$

$m \leftarrow 3$

incrementer1(n, m)

écrire (" n= ", n, " et m= ", m)

**Fin**

**résultat :**

**n=3 et m=4**

Remarque : l'instruction  $x \leftarrow x+1$  n'a pas de sens avec un passage par valeur

# Transmission par valeur, par adresse : exemples

Procédure qui calcule la somme et le produit de deux entiers :

**Procédure** SommeProduit (**x**: entier, **y**: entier, **Var som**: entier, **Var prod**: entier)

    som  $\leftarrow$  x+y

    prod  $\leftarrow$  x\*y

**FinProcédure**

Procédure qui échange le contenu de deux variables :

**Procédure** Echange (**Var x** : réel, **Var y** : réel)

variables    z : réel

    z  $\leftarrow$  x

    x  $\leftarrow$  y

    y  $\leftarrow$  z

**FinProcédure**

# Variables locales et globales (1)

- On peut manipuler 2 types de variables dans un module (procédure ou fonction) : des **variables locales** et des **variables globales**. Elles se distinguent par ce qu'on appelle leur **portée** (leur "champ de définition", leur "durée de vie")
- Une **variable locale** n'est connue qu'à l'intérieur du module où elle a été définie. Elle est créée à l'appel du module et détruite à la fin de son exécution
- Une **variable globale** est connue par l'ensemble des modules et le programme principal. Elle est définie durant toute l'application et peut être utilisée et modifiée par les différents modules du programme

# Variables locales et globales (2)

- La manière de distinguer la déclaration des variables locales et globales diffère selon le langage
  - En général, les variables déclarées à l'intérieur d'une fonction ou procédure sont considérées comme variables locales
- En pseudo-code, on va adopter cette règle pour les variables locales et on déclarera les variables globales dans le programme principal
- **Conseil :** Il faut utiliser autant que possible des variables locales plutôt que des variables globales. Ceci permet d'économiser la mémoire et d'assurer l'indépendance de la procédure ou de la fonction

# Tableaux : saisie et affichage

- Procédures qui permettent de saisir et d'afficher les éléments d'un tableau :

**Procédure** SaisieTab(n : entier, Var T : **tableau** [1..100] de réels )

variable i: entier

**Pour** i de 1 à n Faire

    écrire ("Saisie de l'élément ", i )

    lire (T[i] )

**FinPour**

**Fin Procédure**

**Procédure** AfficheTab(n : entier, T: **tableau** [1..100] de réels)

variable i: entier

**Pour** i de 1 à n Faire

    écrire ("T[",i, "] =", T[i])

**FinPour**

**Fin Procédure**

# Tableaux : exemples d'appel

- Algorithme principal où on fait l'appel des procédures SaisieTab et AfficheTab :

## **Algorithme Tableaux**

variable p : entier

A: **tableau** [1..100] de réels

### **Début**

$p \leftarrow 10$

SaisieTab(p, A)

AfficheTab(10,A)

### **Fin**

# **Exemples : lecture d'une matrice**

- Procédure qui permet de saisir les éléments d'une matrice :

**Procédure** SaisieMatrice(*n* : entier , *m* : entier ,**Var** *A* : **tableau** [1..50, 1..50] de réels)

**Début**

variables *i,j* : entier

**Pour** *i* de 1 à *n* **Faire**

    écrire ("saisie de la ligne ", *i* )

**Pour** *j* de 1 à *m* **Faire**

        écrire ("Entrez l'élément de la ligne ", *i*, " et de la colonne ", *j*)

        lire (*A[i,j]*)

**FinPour**

**FinPour**

**Fin Procédure**

# **Exemples : affichage d'une matrice**

- Procédure qui permet d'afficher les éléments d'une matrice :

**Procédure** AfficheMatrice(n : entier, m : entier ,A : tableau [1..50, 1..50] de réels)

**Début**

variables i,j : entier

**Pour** i de 1 à n **Faire**

**Pour** j de 1 à m **Faire**

    écrire ("A[",i, "] [",j,"]=", A[i,j])

**FinPour**

**FinPour**

**Fin Procédure**

# Exemples : somme de deux matrices

- Procédure qui calcule la somme de deux matrices :

**Procédure** SommeMatrices(n: entier, m : entier,

**A: tableau** [1..50, 1..50] de réels , **B: tableau** [1..50, 1..50] de réels,  
**Var C :tableau** [1..50, 1..50] de réels)

**Début**

variables i,j : entier

**Pour** i de 1 à n **Faire**

**Pour** j de 1 à m **Faire**

C[i,j]  $\leftarrow$  A[i,j]+B[i,j]

**FinPour**

**FinPour**

**Fin Procédure**

# Appel des procédures définies sur les matrices

Exemple d'algorithme principal où on fait l'appel des procédures définies précédemment pour la saisie, l'affichage et la somme des matrices :

## **Algorithme Matrices**

variables M1, M2, M3: **tableau** [1..50,1..50] de réels

**Début**

SaisieMatrice(3, 4, M1)

SaisieMatrice(3, 4, M2)

AfficheMatrice(3,4, M1)

AfficheMatrice(3,4, M2)

SommeMatrice(3, 4, M1,M2,M3)

AfficheMatrice(3,4, M3)

**Fin**

# ***ALGORITHMIQUE***

***Les enregistrements***

# Définition et vocabulaire

- Agrégat d'informations associées à une entité
- **Type complexe** construit à l'aide de type simples ou d'autres types complexes
- Chacune des informations contenue dans une structure s'appelle un **champ**
- Une variable de type structure est aussi appelée un **enregistrement**
  - Analogie avec les bases de données

# Déclaration

---

**Type**

nom = **Structure**

Champ<sub>1</sub> : Type<sub>1</sub>

Champ<sub>2</sub> : Type<sub>2</sub>

...

Champ<sub>n</sub> : Type<sub>n</sub>

**Fin Structure**

# Exemple: Problème

Résoudre le problème suivant:

Afficher les notes d'une promotion par ordre croissant avec le nom et le prénom de chaque étudiant.

Il faut :

Utiliser trois tableaux pour stocker les noms, les prénoms et les notes.

Noms	Prénoms	notes
Derbel	Ali	12
Lajmi	Salah	8
Ayed	Imen	5
Dabech	Sonia	10

Problème :

Lorsque on va trier le tableau des notes, il faut modifier les tableaux noms et prénoms.

# **Exemple: Solution**

## **Utilisation des enregistrements**

C'est intéressant qu'un étudiant soit caractérisé par un type nommé par exemple Etudiant.

**Type**

**Date = Structure**

jour : entier

mois : entier

annee : entier

**Fin Structure**

**Etudiant = Structure**

nom : chaîne de caractères

prenom : chaîne de caractères

datenaiss : Date

note : réel

**Fin Structure**

# Utilisation des structures (1)

Pour remplir une variable de type structure, il faut procéder champ par champ (pas de remplissage global) car les types des champs sont différents

Pour accéder aux attributs d'une variable de type structure, on suffixe le nom de la variable par un « . » suivi du nom de l'attribut

## Exemple

E1, E2: Etudiant

E1.nom: représente le nom de l'étudiant E1

E2.prenom: représente le prénom de l'étudiant E2

E1.datenaiss.jour: représente le jour de naissance de l'étudiant E1

E2.datenaiss.annee: représente l'année de naissance de l'étudiant E2

# Utilisation des structures (2)

- Il est possible de passer un enregistrement en paramètre d'une fonction ou d'une procédure.
- Exemple :

Ecrire une fonction qui retourne la différence de notes entre deux étudiants.

**Fonction différence (e1 : Structure Etudiant, e2 : Structure Etudiant) : entier**

Si (e1.note > e2.note) alors

    Retourner (e1.note - e2.note)

Sinon

    Retourner (e2.note - e1.note)

FinSi

**FinFonction**

# Utilisation des structures (3)

- Une fonction peut retourner une structure
- Une structure peut faire l'objet d'une affectation (avec une variable de même type !)

Exemple:

**e1,e2: Structure Etudiant**

**e2=e1;**

- Les tableaux de structures sont possibles

# TABLEAUX DE STRUCTURES

**Exemple:**

Définir un tableau de 5 étudiants en se référant à la structure Etudiant déjà définie.

1	2	3	4	5
nom	nom	nom	nom	nom
prenom	prenom	prenom	prenom	prenom
datenaiss jour mois annee	datenaiss jour mois annee	datenaiss jour mois annee	datenaiss jour mois annee	datenaiss jour mois annee
note	note	note	note	note

# TABLEAUX DE STRUCTURES

- **Déclaration :**

**Etudiants : tableau [1..5] de Structure Etudiant**

- **Etudiants [1]** représente l'étudiant n° 1
- Pour accéder aux attributs d'une variable de type structure, on suffixe le nom de la variable par un « . » suivi du nom de l'attribut.
- **Exemple :**

Pour accéder au nom du deuxième étudiant il faut écrire :

**Etudiants[2].nom**

# Exercice (1)

On souhaite créer un algorithme qui saisit et affiche les employés d'une société. Chaque employé est défini par : son nom, son prénom, sa date de naissance, son numéro de carte d'identité, sa fonction, son salaire et sa date de recrutement.

1. Définir les structures Employe et Date
2. Ecrire une procédure Saisir\_E qui permet de saisir un employé.
3. Ecrire une procédure Affiche\_E qui affiche un employé.
4. Ecrire une procédure Saisir\_PE qui permet de saisir 50 employés en utilisant la procédure Saisir\_E
5. Ecrire un algorithme principal qui permet de saisir 50 employés et les afficher.

# Exercice (2)

1)

**Type**

**Date = Structure**

jour : entier

mois : entier

annee : entier

**Fin Structure**

**Employe = Structure**

nom : chaîne de caractères

prenom : chaîne de caractères

datenaiss : Date

cin : entier

fonction : chaîne de caractères

salaire : entier

daterec : Date

**Fin Structure**

# **Exercice (3)**

---

**2)**

## **Procédure Saisir\_E (*Var E: Structure Employé*)**

Ecrire ("donner le nom et le prénom de l'employé")

Lire (E.nom, E.prenom)

Ecrire ("donner la date de naissance")

Lire (E.datenaiss.jour, E.datenaiss.mois, E.datenaiss.annee)

Ecrire ("donner le CIN la fonction et le salaire de l'employé")

Lire (E.cin, E.fonction, E.salaire)

Ecrire ("donner la date de recrutement")

Lire (E.daterec.jour, E.daterec.mois, E.daterec.annee)

**Fin Procédure**

# **Exercice (4)**

---

**3)**

## **Procédure Afficher\_E (E : Structure Employé)**

Ecrire ("le nom et le prénom de l'employé : " , E.nom, E.prenom)

Ecrire ("la date de naissance : ", E.datenaiss.jour, " / ", E.datenaiss.mois,  
" / ", E.datenaiss.annee)

Ecrire ("le CIN : " , E.cin, " la fonction : ", E.fonction," le salaire : ",  
E.salaire)

Ecrire (" la date de recrutement : ", E.daterec.jour, " / ", E.daterec.mois,  
" / ", E.daterec.annee )

**Fin Procédure**

# Exercice (5)

4)

**Procédure Saisir\_PE (Var Emp: tableau[1..50] de Structure Employe)**

**Variables** i: entier

**Pour** i de 1 à 50 **Faire**

Ecrire ("donner le nom et le prénom de l'employé numéro", i)

Saisir\_E (Emp[i])

**Fin Pour**

**Fin Procédure**

# **Exercice (6)**

---

**5)**

**Algorithme Exercice\_Structures**

**Variables** i: entier

Emp: tableau [1..50] de structure Employe

**Début**

Saisir\_PE(Emp)

**Pour** i **de** 1 **à** 50 **Faire**

Afficher\_E(Emp[i])

**Fin Pour**

**Fin**

# ***ALGORITHMIQUE***

***Algorithmes de tri des tableaux***

# Algorithmes de tri des tableaux

- Les tableaux permettent de stocker plusieurs éléments de même type au sein d'une seule entité,
- Lorsque le type de ces éléments possède un ordre total, on peut donc les ranger en ordre croissant ou décroissant,
- Trier un tableau c'est donc ranger les éléments d'un tableau en ordre croissant ou décroissant
- Dans ce cours on ne fera que des tris en ordre croissant
- Il existe plusieurs méthodes de tri qui se différencient par leur complexité d'exécution et leur complexité de compréhension pour le programmeur.

# Tri à Bulles: Principe

- Le tri à bulles est un algorithme de tri classique. Son principe est simple, et il est très facile à implémenter
- L'algorithme parcourt le tableau, et dès que deux éléments consécutifs ne sont pas ordonnés, les échange
- Après un premier passage, on voit que le plus grand élément se situe bien enfin du tableau
- On recommence un tel passage, en s'arrêtant à l'avant-dernier élément, et ainsi de suite
- Au i-ème passage, on fait remonter le i-ème plus grand élément du tableau à sa position définitive, un peu à la manière de bulles qu'on ferait remonter à la surface d'un liquide, d'où le nom d'algorithme de tri à bulles.

# Tri à Bulles: Algorithme

**Procédure** tri\_Bulle (**Var** tab: tableau [1..50] d'entiers, N: entier )

**Variables** i, k, tmp : entier

**Pour** i de N à 2 **Faire**

**Pour** k de 1 à i-1 **Faire**

**Si** (tab[k] > tab[k+1]) **alors**

    tmp  $\leftarrow$  tab[k]

    tab[k]  $\leftarrow$  tab[k+1]

    tab[k+1]  $\leftarrow$  tmp

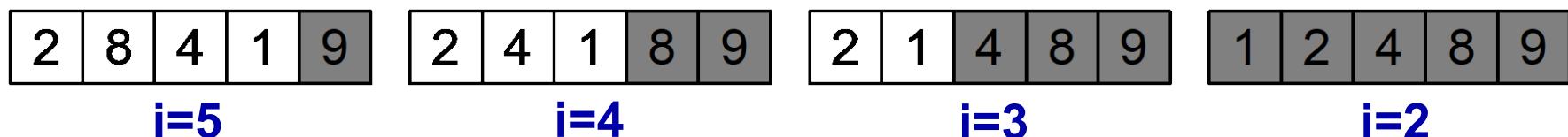
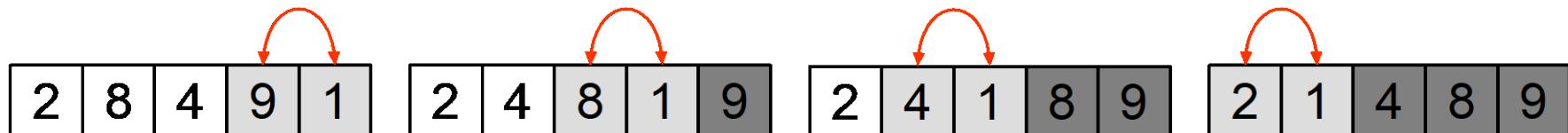
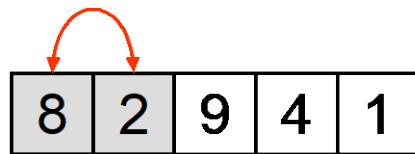
**Fin si**

**Fin pour**

**Fin pour**

**Fin Procédure**

# Tri à Bulles: Exemple



# Tri par insertion: Principe

- Le tri par insertion est également un algorithme de tri classique, simple à implémenter et intuitif, puisqu'il est celui que les joueurs de cartes utilisent naturellement pour trier leurs cartes.
- On parcourt le tableau du début à la fin, et à l'étape  $i$ , on considère que les éléments de 1 à  $i-1$  du tableau sont déjà triés.
- On va alors placer le  $i$ -ème élément à sa bonne place parmi les éléments précédents du tableau, en le faisant redescendre jusqu'à atteindre un élément qui lui est inférieur

# Tri par insertion: Algorithme

**Procédure** tri\_Insertion (**Var** tab: tableau[1..50] d'entiers, N:entier)

**Variables** i, k, tmp :entier

**Pour** i de 2 à N **Faire**

    Tmp  $\leftarrow$  tab[i]

    K  $\leftarrow$  i

**Tant que** (k > 1 ET tab[k - 1] > tmp) **Faire**

        tab[k]  $\leftarrow$  tab[k - 1];

        k  $\leftarrow$  k - 1;

**Fin Tant que**

    tab[k]  $\leftarrow$  tmp;

**Fin pour**

**Fin Procédure**

# Tri par insertion: Exemple

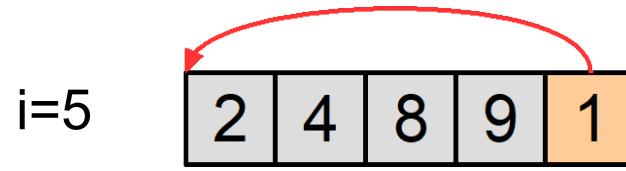
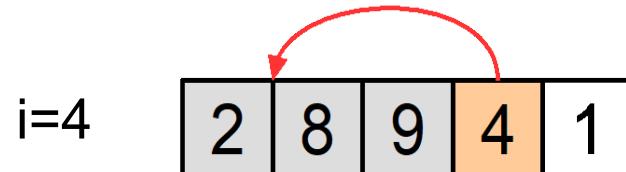
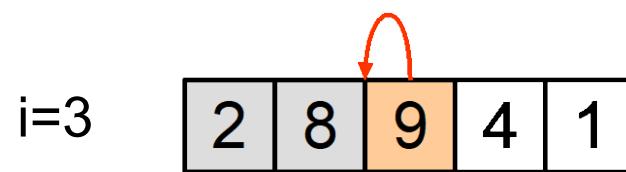
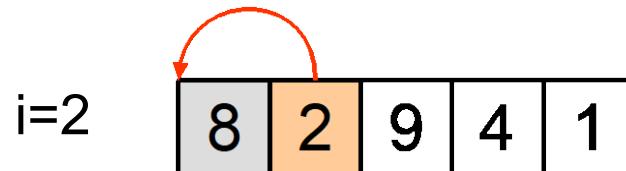


Tableau trié | 

1	2	4	8	9
---	---	---	---	---

# Tri par sélection: Principe

- Le tri par sélection (ou tri par extraction) est un algorithme de tri par comparaison
- Trouver le plus petit élément et le mettre au début du tableau
- Trouver le 2ème plus petit élément et le mettre en seconde position
- Trouver le 3ème plus petit élément et le mettre à la 3e place,
- ...

# Tri par sélection: Algorithme

**Procédure** (**Var** tab: tableau[1..50] d'entiers, N:entier)

**Variables** i, j, tmp, small : entier

**Pour** i de 1 à N-1 **Faire**

    small ← i

**Pour** j de i+1 à N **Faire**

**Si** (tab[j] < tab[small]) **alors**

        small ← j

**Fin si**

**Fin pour**

    tmp ← tab[small]

    tab[small] ← tab[i]

    tab[i] ← tmp

**Fin pour**

**Fin Procédure**

# Tri par sélection: Exemple

8	2	9	4	1
---	---	---	---	---

i=1

1	2	9	4	8
---	---	---	---	---

i=2

1	2	9	4	8
---	---	---	---	---

i=3

1	2	4	9	8
---	---	---	---	---

i=4

1	2	4	8	9
---	---	---	---	---

# ***ALGORITHMIQUE***

***Algorithmes de recherche***

# Recherche séquentielle

- Recherche de la valeur x dans un tableau T de N éléments :

**Variables** i: entier, Trouve : booléen

...

i $\leftarrow$ 1 , Trouve  $\leftarrow$  Faux

**TantQue** (i  $\leq$  N) ET (Trouve=Faux) **Faire**

**Si** (T[i]=x) **alors**

    Trouve  $\leftarrow$  Vrai

**Sinon**

    i $\leftarrow$ i+1

**FinSi**

**FinTantQue**

**Si** Trouve **alors** // c'est équivalent à écrire **Si** (Trouve=Vrai) **alors**

    écrire ("x appartient au tableau")

**Sinon**

    écrire ("x n'appartient pas au tableau")

**FinSi**

## **Recherche séquentielle (version 2)**

- Une fonction Recherche qui retourne un booléen pour indiquer si une valeur x appartient à un tableau T de dimension N.  
x , N et T sont des paramètres de la fonction

**Fonction** Recherche(x : réel, N: entier, T:tableau[1..100] de réels ) : booléen

**Variable** i: entier

**Pour** i de 1 à N **Faire**

**Si** (T[i]=x) **alors**

    retourner (Vrai)

**FinSi**

**FinPour**

    retourner (Faux)

**FinFonction**

# Recherche dichotomique

- Dans le cas où le tableau est ordonné, on peut améliorer l'efficacité de la recherche en utilisant la méthode de recherche dichotomique
- **Principe :** diviser par 2 le nombre d'éléments dans lesquels on cherche la valeur  $x$  à chaque étape de la recherche. Pour cela on compare  $x$  avec  $T[\text{milieu}]$  :
  - Si  $x < T[\text{milieu}]$ , il suffit de chercher  $x$  dans la 1ère moitié du tableau entre ( $T[0]$  et  $T[\text{milieu}-1]$ )
  - Si  $x > T[\text{milieu}]$ , il suffit de chercher  $x$  dans la 2ème moitié du tableau entre ( $T[\text{milieu}+1]$  et  $T[N-1]$ )

# Recherche dichotomique : algorithme

inf←1 , sup←N, Trouve ← Faux

**TantQue** (inf <=sup) ET (Trouve=Faux) **Faire**

    milieu←(inf+sup) div 2

**Si** (x=T[milieu]) **alors**

        Trouve ← Vrai

**Sinon Si** (x>T[milieu]) **alors**

        inf←milieu+1

**Sinon** sup←milieu-1

**FinSi**

**FinSi**

**FinTantQue**

**Si** Trouve **alors** écrire ("x appartient au tableau")

**Sinon** écrire ("x n'appartient pas au tableau")

**FinSi**

# ***ALGORITHMIQUE***

***La récursivité***

# Définitions

---

- Un algorithme est dit récursif s'il est défini en fonction de lui-même.
  - La récursion est un principe puissant permettant de définir une entité à l'aide d'une partie de celle-ci.
- Chaque appel successif travaille sur un ensemble d'entrées toujours plus affinée, en se rapprochant de plus en plus de la solution d'un problème.

# Evolution d'un appel récursif

L'exécution d'un appel récursif passe par deux phases, la phase de descente et la phase de la remontée :

- Dans la phase de descente, chaque appel récursif fait à son tour un appel récursif. Cette phase se termine lorsque l'un des appels atteint une condition terminale.  
→ condition pour laquelle la fonction doit retourner une valeur au lieu de faire un autre appel récursif.
- Ensuite, on commence la phase de la remontée. Cette phase se poursuit jusqu'à ce que l'appel initial soit terminé, ce qui termine le processus récursif.

# Types de récursivité

**La récursivité simple:** une récursivité est simple si on fait un seul appel récursif pour la fonction P dans le corps d'une fonction recursive P.

**Exemple:**

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n + 1) & \text{si } n > 0 \end{cases}$$

**La récursivité multiple:** une récursivité est multiple s'il y a plusieurs appels récursifs à une fonction P dans le corps d'une fonction recursive P.

**Exemple:**

$$C_n^P = \begin{cases} 1 & \text{si } p = 0 \\ 1 & \text{si } n = 0 \\ C_{n-1}^P + C_{n-1}^{P-1} & \text{sinon} \end{cases}$$

# Types de récursivité

**La récursivité imbriquée:** Une récursivité est dite imbriquée si une fonction récursive P contient un appel imbriqué.

**Exemple:**

$$Ack(n, p) = \begin{cases} p + 1 & \text{si } n = 0 \\ Ack(n - 1, 1) & \text{si } n > 0 \text{ et } p = 0 \\ Ack(n - 1, Ack(n, p - 1)) & \text{sinon} \end{cases}$$

**La récursivité mutuelle:** Une récursivité est mutuelle ou croisée quand une fonction P appelle une autre fonction Q qui déclenche un appel récursif à P

**Exemple:**

$$pair(n) = \begin{cases} \text{vrai} & \text{si } n = 0 \\ \text{impair}(n - 1) & \text{sinon} \end{cases}$$

$$impair(n) = \begin{cases} \text{faux} & \text{si } n = 0 \\ pair(n - 1) & \text{sinon} \end{cases}$$

# Récursivité non terminale (1)

Une fonction récursive est dite **non terminale** si le résultat de l'appel récursif est utilisé pour réaliser un traitement

- ➔ L'appel n'est pas la dernière instruction et/ou
- ➔ fait partie d'une expression

# Récurseurité non terminale (2)

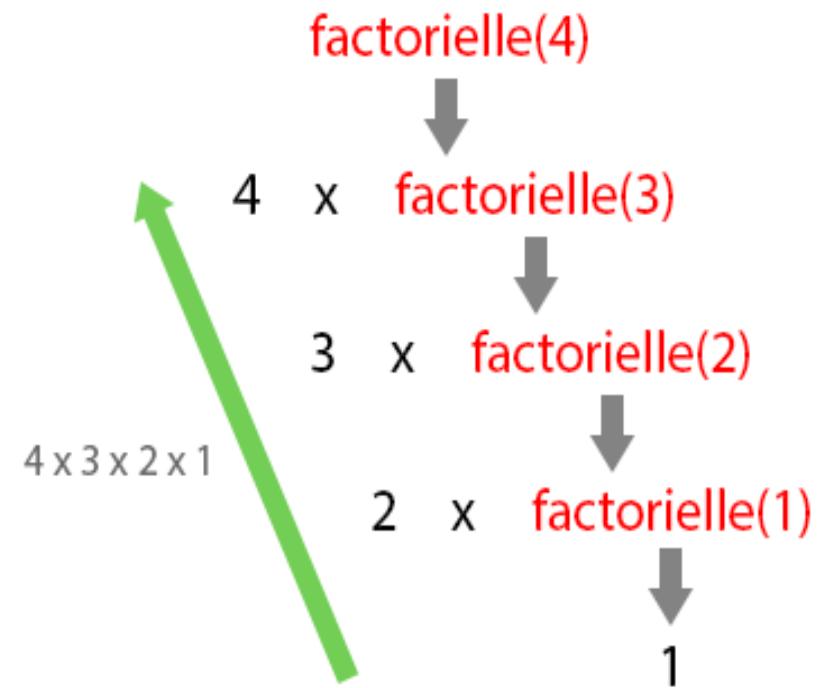
Exemple : Factorielle de n

$$0! = 1! = 1$$

$$n! = n \times (n-1)!$$

$$\begin{aligned} 4! &= 4 \times 3! = 4 \times 3 \times 2! = 4 \times \\ &\quad \times 2 \times 1! \end{aligned}$$

$$= 4 \times 3 \times 2 \times 1$$



# Récurseurité non terminale (3)

- Règle de récursion : (calcul de la factorielle)

Décrémenter la valeur de n à chaque appel de la fonction jusqu'à ce que  $n=1$

- Règle de sortie de la fonction récursive :  
si  $n = 1$  c'est la fin de la récursion.

## Remarque:

- Il faut faire attention au choix de la condition de sortie. Il faut être sûr que cette condition est valide sinon on se ramène à une boucle infinie sans condition de sortie.

# Récurseurité non terminale (5)

**Fonction Factorielle (n : entier) : entier**

Si ( $n = 1$  ou  $n= 0$  ) alors

    retourner 1

Sinon

    retourner  $n \times$  **Factorielle (n-1)**

Finsi

**FinFonction**

# Récurseurité non terminale (6)

Trace d'exécution de la fonction factorielle (calcul de la valeur de 4!)

Factorielle (4) = 4 \* Factorielle (3)

phase de descente

Factorielle (3) = 3 \* Factorielle (2)

Factorielle (2) = 2 \* Factorielle (1)

Factorielle (1) = 1

condition terminale

Factorielle (2) = 2 \* 1

phase de remontée

Factorielle (3) = 3 \* 2

Factorielle (4) = 4 \* 6

récursion terminée



## Récursivité non terminale (7)

- Lorsque la fonction rencontre la condition de sortie, elle remonte dans tous les appels précédents pour calculer n avec la valeur précédemment trouvée.
- Les appels des fonctions récursives sont empilés : pile LIFO : Last In First Out Dernier Entré Premier Sorti.
- Chaque appel se trouve l'un à la suite de l'autre dans la pile du programme.
- Une fonction de ce type possède deux parcours : la phase de descente et la phase de remontée.
- Dans la phase de remontée, les appels enregistrés sont dépiler au fur et à mesure de la remontée.

## Récursivité non terminale (8)

Une approche récursive non terminale demande beaucoup de moyens en ressources car énormément de données doivent être stockées dans la pile d'exécution.

- Risque de faire exploser la pile.
- Plantage du programme.

# Récursivité terminale (1)

- Une fonction récursive est dite récursive **terminale** si tous ses appels sont récursifs terminaux.
- Un appel récursif est terminal s'il s'agit de la dernière instruction exécutée dans le corps d'une fonction et que sa valeur de retour ne fait pas partie d'une expression.
- Les fonctions récursives terminales sont caractérisées par le fait qu'elles n'ont rien à faire pendant la phase de remontée

# Récursivité terminale (2)

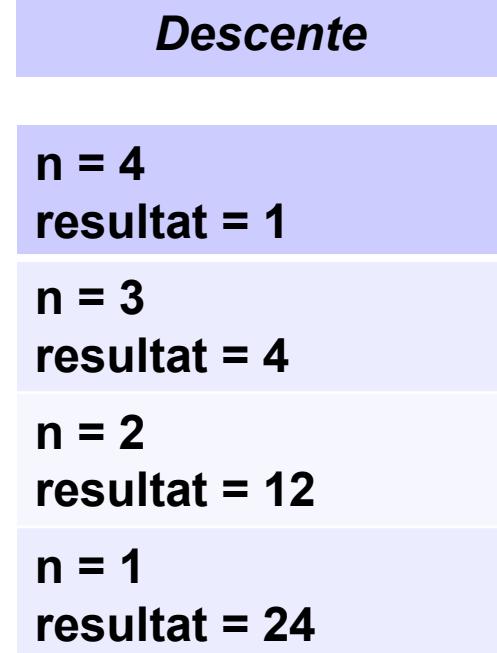
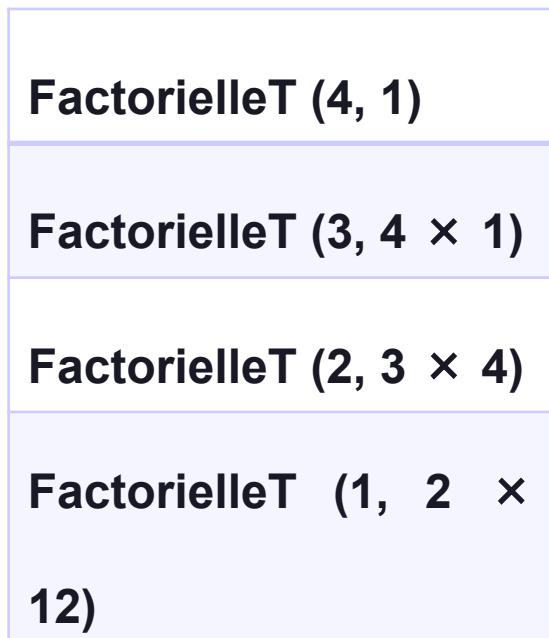
Argument supplémentaire pour créer une récursivité terminale

**Fonction FactorielleT (n : entier, résultat : entier) : entier**

Si (n = 1 ou n= 0 ) alors  
    retourner résultat  
sinon  
    retourner **FactorielleT (n-1 , n × résultat)**  
Finsi  
**FinFonction**

# Récurseurité terminale (3)

Trace d'exécution de la fonction FactorielleT (calcul de la valeur de 4!)



Cette fonction possède une seule **phase de descente**  
=> Gain du temps d'exécution  
=> Moins de risque de plantage

Condition de sortie: n = 1  
Retour de la fonction: 24

# Exercice (1)

Ecrire la fonction Somme qui retourne la somme de n premiers nombres.

## Version Non Récursive :

**Fonction Somme (n : entier) : entier**

**Variable**

r, i : entier

r <- 0

**Pour i de 1 à n Faire**

    r <- r + i

**Fin Pour**

retourner r

**FinFonction**

**Algorithme Principal**

**Début**

    écrire ("la somme des 5 premiers nombre est : " Somme(5));

**Fin**

# Exercice (2)

**Version Récursive non terminale :**

**Fonction Somme (n : entier) : entier**

Si ( $n \leq 1$ ) alors

    retourner n

Sinon

    retourner ( $n + \text{somme}(n-1)$ )

FinSi

**FinFonction**

# Exercice (3)

**Version Récursive terminale :**

**Fonction Somme (n : entier, r : entier) : entier**

Si ( $n < 1$ ) alors

    retourner r

Sinon

    retourner (somme( $n-1$ ,  $n + r$ ))

FinSi

**FinFonction**

# *Algorithmique*

*Les pointeurs*

# **Importance des pointeurs**

- On peut accéder aux données en mémoire à l'aide de pointeurs i.e. des variables pouvant contenir des adresses d'autres variables.
- En C, les pointeurs jouent un rôle primordial dans la définition de fonctions :
  - ➔ Les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions.
- Les pointeurs nous permettent de définir de nouveaux types de données : les piles, les files, les listes, ....
- Les pointeurs nous permettent d'écrire des programmes plus compacts et plus efficaces.

# Notion du pointeur

- Une variable qui contient **l'adresse** d'une autre variable
- Un pointeur est souvent typé (rarement générique)
- Il peut se déplacer d'une case mémoire à une autre rapidement
  - Manipulation directe de la mémoire
  - Utilisation de la mémoire dynamique

# Syntaxe

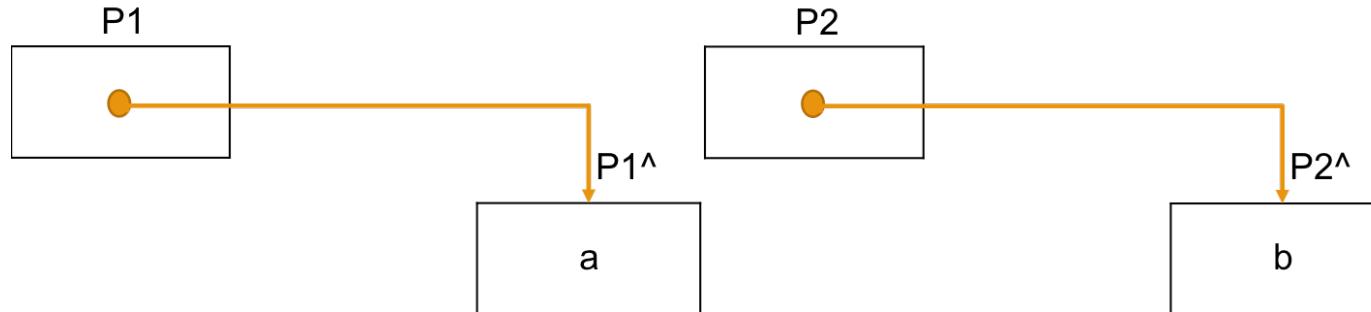
- **Déclaration**
  - Nom\_Pointeur:<sup>^</sup>Type  
(Type: type de données vers lequel il peut pointer )
- **Attribution d'une adresse ou d'un objet pointé**  
Nom\_Pointeur $\leftarrow$ @X (avec x de même type que P)
- **Accès à la valeur de l'objet pointé**
  - Nom\_pointeur<sup>^</sup>

# Gestion des pointeurs

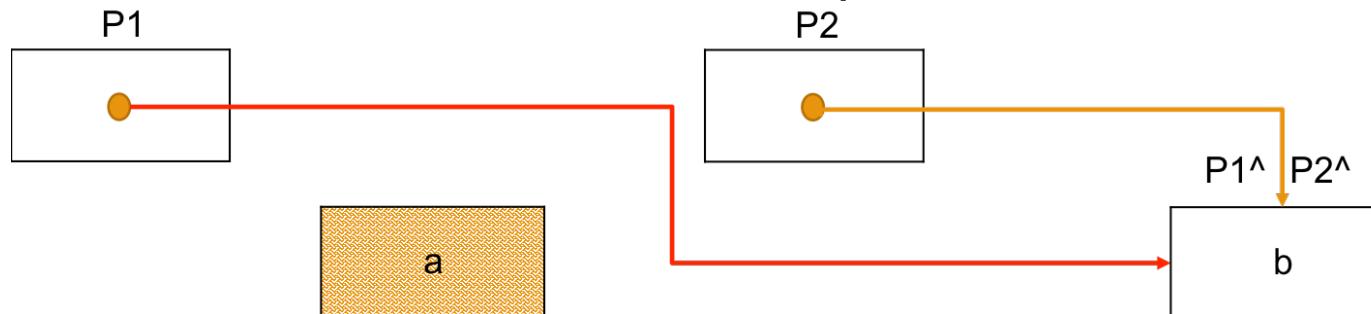
- **Création du pointeur**
  - Déclaration et assignation d'un type  
Nom\_Pointeur:<sup>^</sup>type
- **Initialisation** (un pointeur non initialisé peut pointer n'importe où)
  - Vers une donnée valide et statique de même type
  - En lui assignant la valeur « Null »
  - En allouant dynamiquement une zone mémoire vers laquelle le pointeur va pointer
- **Destruction du pointeur**
  - Appel d'une fonction prédéfinie → La valeur pointée est détruite

## Exemple

Soient deux variables  $P1$  et  $P2$  de type pointeur permettant l'accès respectivement à  $a$  et  $b$ .



L'exécution de l'instruction  $P1 \leftarrow P2$  ; permet d'obtenir :

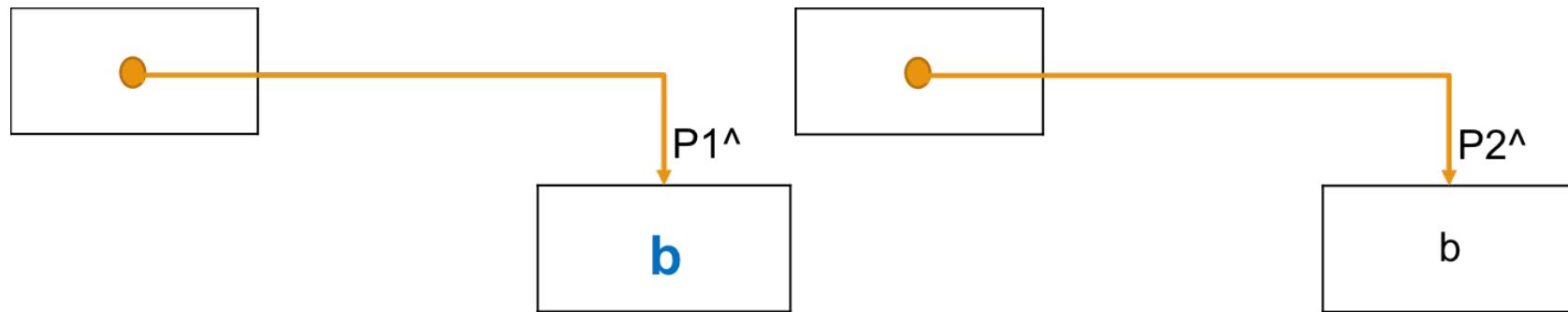


→  $a$  n'est pas accessible, car on a perdu son adresse qui se trouvait dans  $P1$

→ On peut atteindre  $b$  par l'intermédiaire de  $P1$  ou de  $P2$ .

## **Exemple...**

L'exécution de l'instruction  $P1^{\wedge} \leftarrow P2^{\wedge}$  permet d'obtenir :



- *a* n'existe plus (précédemment, il existait toujours, mais n'était plus accessible).
- *b* existe à deux exemplaires; l'un des exemplaires à une adresse dont la valeur est dans *P1* et l'autre à une adresse différente dont la valeur est dans *P2*.

# Pointeur et tableau

- Tout tableau en C est en fait un pointeur constant

**Exemple:** int tab[10]; tab est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau.

- tab contient l'adresse de tab[1]
- $\text{tab}^\wedge$  contient la valeur de tab[1]
- $(\text{tab}+1)^\wedge$  contient la valeur de tab[2]
- **p[i]  $\Leftrightarrow$  (p + i - 1)^\wedge**
- On utilise les pointeurs pour manipuler des tableaux dont on ne connaît pas la taille à la compilation

# Pointeur et tableau: Exemple

Ecrire un algorithme qui copie les éléments positifs d'un tableau T dans un tableau POS.

**Sans pointeur:**

**Algorithme** TableauV1

**Type** : Tab : Tableau [1..50] d'entiers

**Variable** : T, POS : Tab

n, i, j : entier

**Début**

**Répéter**

Ecrire ("Donner la taille du tableau")

Lire (n)

**Jusqu'à** (n $\geq$ 1 et n $\leq$ 50)

**Pour** i de 1 à n Faire

Ecrire ("Donner l'élément n° ", i)

Lire (T[i])

**Fin pour**

j  $\leftarrow$  1

**Pour** i de 1 à n Faire

**Si** (T[i] > 0) **alors**

POS[j]  $\leftarrow$  T[i]

J  $\leftarrow$  j + 1

**FinSi**

**Fin pour**

**Pour** i de 1 à j-1 Faire

Ecrire (POS[i])

**Fin pour**

**Fin**

# Pointeur et tableau: Exemple...

Avec pointeur:

**Algorithme** TableauV2

**Type** : Tab : Tableau [1..50] d'entiers

**Variables** : T, POS : Tab

p1, p2 : pointeur sur entier

**Début**

**Répéter**

Ecrire ("Donner la taille du tableau")

Lire (n)

**Jusqu'à** (n>=1 et n<=50)

**Pour** p1 de T à T+n-1 **Faire**

Ecrire ("Donner l'élément n° ", p1-T+1)

Lire (p1<sup>^</sup>)

**Fin pour**

p2  $\leftarrow$  POS

**Pour** p1 de T à T+n-1 **Faire**

**Si** (p1<sup>^</sup> > 0) **alors**

p2<sup>^</sup>  $\leftarrow$  p1<sup>^</sup>

p2  $\leftarrow$  p2 + 1

**FinSi**

**Fin pour**

**Pour** p1 de POS à p2-1 **Faire**

Ecrire (p1<sup>^</sup>)

**Fin pour**

**Fin**

# **Pointeur et chaînes de caractères**

- Tout ce qui a été mentionné concernant les pointeurs et les tableaux reste vrai pour les pointeurs et les chaînes de caractères.  
→ Une chaîne de caractères est traitée comme un tableau à une dimension de caractères, se terminant par le caractère nul \0.
- On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur.
- Cela est souvent utilisé car on connaît rarement à la compilation les tailles de chaînes dont on va avoir besoin.

# **Pointeur et chaînes de caractères: Exemple**

Ecrire un algorithme qui lit une chaîne de caractères CH et détermine la longueur de la chaîne. (utiliser les pointeurs)

**Algorithme** LongCH

**Variables** : CH[100] : Chaîne de caractères

p : pointeur sur chaîne de caractères

**Début**

Ecrire ("Donner une chaîne de caractères")

Lire (CH)

p  $\leftarrow$  ch

**Tant que** ( $p^{\wedge} <> \backslash 0$ ) **Faire**

    p  $\leftarrow$  p+1

**Fin tant que**

Ecrire ("la longueur de CH est ", p-CH)

**Fin**

# Les pointeurs en C

- **Déclaration**

```
type de donnée * identificateur de variable pointeur;
```

**Exemple :** `int * pNombre;`

`pNombre` désigne une variable pointeur pouvant contenir uniquement l'adresse d'une variable de type `int`.

- Les pointeurs et les noms de variables ont le même rôle : ils donnent accès à un emplacement en mémoire.
- Par contre, un pointeur peut contenir différentes adresses mais le nom d'une variable (pointeur ou non) reste toujours lié à la même adresse.

# Les pointeurs en C...

- Pour obtenir l' adresse d' une variable, on utilise l' opérateur **&** précédant le nom de la variable.
- **Syntaxe:** & nom de la variable

Ex1 : int A;

int \* pNombre = &A;

ou encore

int \* pNombre;

pNombre = &A;

Ex2. : int N;

printf("Entrez un nombre entier : ");

scanf("%d", &N);

scanf a besoin de l' adresse de chaque paramètre pour pouvoir lui attribuer une nouvelle valeur.

**L' opérateur & ne peut être appliqué ni à des constantes ni à des expressions.**

# Les pointeurs en C...

- Pour avoir accès au contenu d'une adresse, on utilise l'opérateur \* précédant le nom du pointeur.

Ex. :

```
int A = 10, B = 50;  
int * P;
```

```
P = &A;  
B = *P;  
*P = 20;
```

