

# ALGORITHME & PROGRAMMATION STRUCTURÉE

Marine ABADI / UTOPIOS / M2I Formation

# Objectif de ce module

## ■ Indispensable avant la programmation

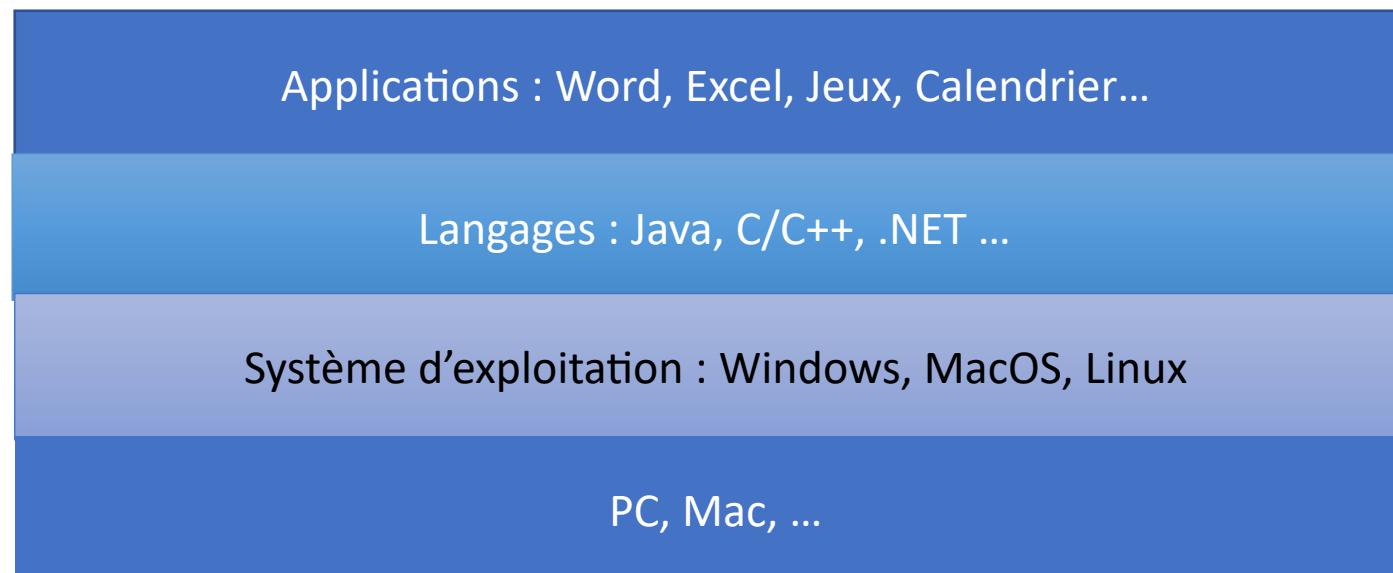
- ➔ Apprendre les concepts de base de l'algorithme et de la programmation
- ➔ Etre capable de mettre en œuvre ces concepts pour analyser des problèmes simples et écrire les programmes correspondants

# Pourquoi fait-on des algorithmes ?

# Généralités en informatique

## L'informatique : c'est quoi ?

- Techniques du traitement **automatique** de l'**information** au moyen des ordinateurs
- Eléments d'un système informatique



# Généralités en informatique

## ■ Les éléments d'un ordinateur

### → Unité centrale (le boîtier)

- Processeur ou CPU (*Central Processing Unit*)
- Mémoire centrale
- Disque dur, lecteur ...
- Cartes spécialisées (cartes graphiques, réseau, ...)
- Interfaces d'entrée-sortie (USB, HDMI, VGA)

### → Périphériques

- Moniteur, clavier, souris
- Modem, imprimante, scanner, ...

# Généralités en informatique

## Qu'est-ce qu'un système d'exploitation ?

- ➡ Ensemble de programmes qui gère le matériel et contrôle les applications :
  - **Gestion des périphériques**  
Affichage à l'écran, lecture du clavier, pilotage d'une imprimante, ...
  - **Gestion des utilisateurs et de leurs données**  
Comptes, partage des ressources, gestion des fichiers et répertoires, ...
  - **Interface avec l'utilisateur**  
Textuelle ou graphique : Interprétation des commandes
  - **Contrôle des programmes**  
Découpage en tâches, partage du temps processeur, ...

# Généralités en informatique

## ■ Qu'est-ce qu'un langage informatique ?

- ➡ Un langage informatique est un outil permettant de donner des ordres (**instructions**) à la machine à chaque instruction correspond une action du processeur.
- ➡ **Intérêt** : écrire des programmes (suite consécutive d'instructions) destinés à effectuer une tache donnée
- ➡ **Exemple** : un programme de gestion de comptes bancaires
- ➡ **Contrainte** : être compréhensible par la machine

# Généralités en informatique

## ■ Qu'est-ce que le langage machine ?

- ➔ Langage **binnaire**: l'information est exprimée et manipulée sous forme d'une suite de bits
- ➔ Un **bit (binary digit)** = 0 ou 1 (2 états électriques)
- ➔ Une combinaison de 8 bits = 1 **Octet (byte)** => $2^8= 256$  possibilités qui permettent de coder tous les caractères alphabétiques, numériques, et symboles tels que ?, \*, &, ...
  - Le code **ASCII** (*American Standard Code for Information Interchange*) donne les correspondances entre les caractères alphanumériques et leurs représentation binaire, Ex. A= 01000001 etc...
- ➔ Les opérations logiques et arithmétiques de base (addition, multiplication, ...) sont effectuées en binaire

# Généralités en informatique

## L'assembleur

- ➡ Problème : le langage machine est difficile à comprendre par l'humain
  - ➡ Idée : trouver un langage compréhensible par l'homme qui sera ensuite converti en langage machine
- Assembleur** (1er langage) : exprimer les instructions élémentaires de façon symbolique



- + : déjà plus accessible que le langage machine
- - : dépend du type de la machine (n'est pas **portable**)
- - : pas assez efficace pour développer des applications complexes



**Apparition des langages évolutés**

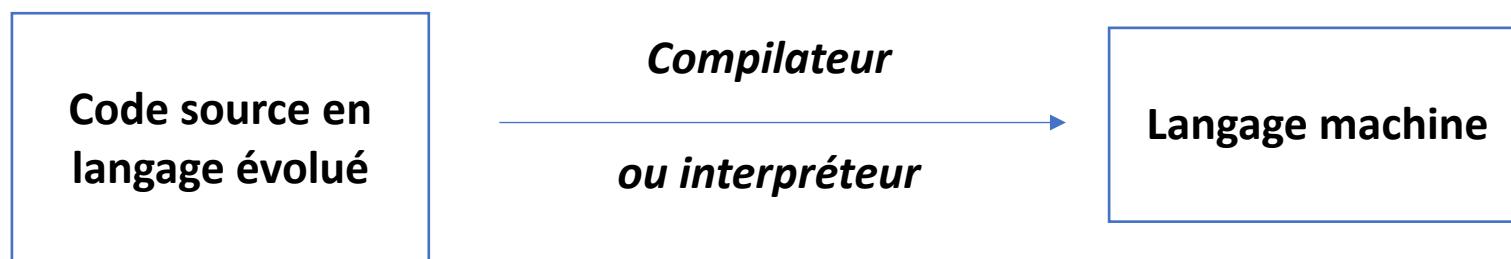
# Généralités en informatique

## ■ Langage haut niveau

### → Intérêts multiples pour le haut niveau :

- proche du langage humain «anglais» (compréhensible)
- permet une plus grande portabilité (indépendant du matériel)
- Manipulation de données et d'expressions complexes (réels, objets,  $a^*b/c$ , ...)

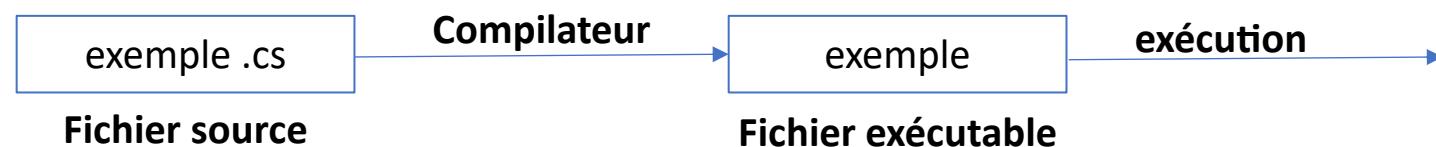
→ Nécessité d'un **traducteur** (compilateur/interpréteur), exécution plus ou moins lente selon le traducteur



# Généralités en informatique

## Compilateur

→ **Compilateur** : traduire le programme entier une fois pour toutes



- + plus rapide à l'exécution
- + sécurité du code source
- - il faut recompiler à chaque modification

# Généralités en informatique

## ■ Interpréteur

➡ **Interpréteur** : traduire au fur et à mesure les instructions du programme à chaque exécution



- + exécution instantanée appréciable pour les débutants
- - exécution lente par rapport à la compilation

# Généralités en informatique

## ■ Un programme

- ➔ Un programme correspond à une méthode de résolution pour un problème donné.
- ➔ Cette méthode de résolution est effectuée par une suite d'instructions d'un langage de programmation
- ➔ Ces instructions permettent de traiter et de transformer les **données** (entrées) du problème à résoudre pour aboutir à des résultats (sorties)
- ➔ Un programme n'est pas une solution en soi mais une **méthode à suivre** pour trouver des solutions.
- ➔ La programmation est l'ensemble des activités orientées vers la conception, la réalisation, le test et la maintenance de programmes

# Généralités en informatique

## ■ Langages de programmation

### → Types de langages :

- Langages procéduraux
- Langages orientés objets

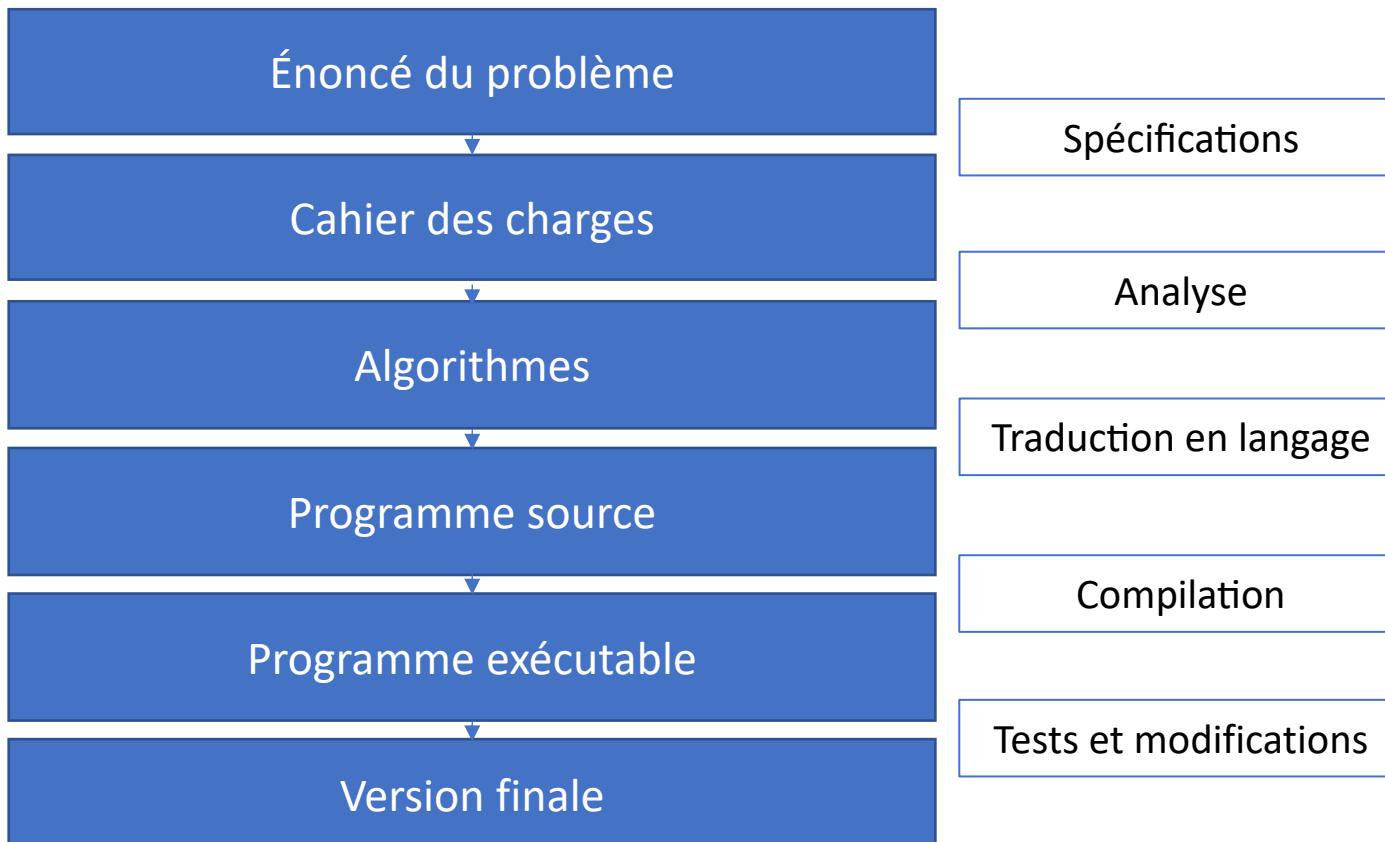
### → Exemple :

- C, PHP
- C++, Java, C# ...

### → Choix d'un langage ?

# Généralités en informatique

## Etapes de la réalisation d'un programme



→ La réalisation de programmes passe par l'**écriture d'algorithmes**

# Généralités en informatique

## L'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), ...
- ➡ Une bonne connaissance de l'algorithmique permet d'écrire des algorithmes **exacts et efficaces**

# Généralités en informatique

## Algorithme et programmation

→ Un algorithme est une suite d'instructions ayant pour but de résoudre un problème donné  
Un algorithme peut se comparer à une recette de cuisine

→ L'élaboration d'un algorithme précède l'étape de programmation

- Un programme est un algorithme
- Un langage de programmation est un langage compris par l'ordinateur
- L'algorithme est la résolution brute d'un problème informatique

# Généralités en informatique

## Représentation d'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

# Notions de bases en algorithmique

# Notions de base

## Notions 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, elle doivent être caractérisées par :
  - un nom (**Identificateur**)
  - un **type** (entier, réel, caractère, chaîne de caractères, ...)

# Notions de base

## Choix des identificateurs

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

- Un nom doit commencer par une lettre alphabétique **exemple valide: A1** **exemple invalide: 1A**
- Il doit être constitué uniquement de lettres, de chiffres et du soulignement \_  
**valides: cdi2016, cdi\_2016** **invalides: cdi 2016,cdi-2016,cdi;2016**
- Il doit être différent des mots réservés du langage  
(par exemple en Java: **int, float, else, switch, case, default, for, main, return**, ...)
- La longueur du nom doit être inférieure à la taille maximale spécifiée par le langage utilisé

# Notions de base

## Choix des identificateurs

- ➡ **Conseil :** pour la lisibilité du code, choisir des noms significatifs qui décrivent les données manipulées  
exemples: `TotalVentes2016`, `Prix_TTC`, `Prix_HT`
  
- ➡ **Remarque :** en pseudo-code algorithmique, on va respecter les règles citées,  
même si on est libre dans la syntaxe

# Notions de base

## Type de variable

→ 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 : toutes suites de caractères **exemples: " Nom, Prénom", "code postal: 1000", ...**

# Notions de base

## Déclaration des variables

→ Toutes variables utilisées 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**

**Vrai / Faux: booléen**

**ch1, ch2 : chaîne de caractères**

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

# Notions de base

## L'instruction d'affectation

→ L'**affectation** consiste à attribuer une valeur à une variable  
c'est le fait de remplir où de modifier le contenu d'une zone mémoire

→ En pseudo-code, l'affectation se note avec le signe ←

Var← 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 ←1                          j ←i                          k ←i+j  
                          x ←10.3                        OK ←FAUX                        ch1 ←"SMI"  
                          ch2 ←ch1                        x ←4                                x ←j

→ non valides : i ←10.3                        OK ←"SMI »                        j ←x

# Notions de base

## ■ Attention

➡ Beaucoup de langages de programmation (C/C++, Java, ...) utilisent le signe égal = pour l'affectation  $\leftarrow$ .

### Ne pas confondre :

- l'affectation n'est pas commutative :  $A=B$  est différent de  $B=A$
- l'affectation est différente d'une équation mathématique :
  - $A=A+1$  a un sens en langage de programmation
  - $A+1=2$  n'est pas possible en langage 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

# Notions de base

EXERCICE

## ■ Exercice 1 (simple sur l'affectation)

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

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

**Début algorithme**

A  $\leftarrow$  3

B  $\leftarrow$  7

A  $\leftarrow$  B (devient 7)

B  $\leftarrow$  A + 5 (devient  $7+5=12$ )

C  $\leftarrow$  A + B (devient  $7+12=19$ )

C  $\leftarrow$  B - A (devient  $12-7=5$ )

**Fin algorithme**

# Notions de base

EXERCICE

## ■ Exercice 2 (simple sur l'affectation)

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

**Variables A, B : Entier**

**Début**

A ← 1

B ← 2

A ← B (devient 2)

B ← A

**Fin**

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

# Notions de base

EXERCICE

## Exercice 3 (simple sur l'affectation)

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

# Notions de base

## ■ 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 : reste de la division euclidienne** ), ^ (**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**

# Notions de base

## Priorités des opérateurs

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

1. ^ élévation à la puissance
2. \* , / multiplication, division
3. % modulo
4. + , - 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**

# Notions de base

## Instructions d'entrée/Sortie : lecture et écriture

- 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

# Notions de base

## Instructions d'entrée/Sortie : lecture et écriture

→ **L'écriture** permet d'afficher des résultats à l'écran (ou de les écrire dans un fichier)

En pseudo-code, on note: **écrire (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

# Lecture & écriture

EXEMPLE

## Instructions d'entrée/Sortie : 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**

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

**Fin**

# Lecture & écriture

EXAMPLE

## Instructions d'entrée/Sortie : lecture et écriture

→ Ecrire un algorithme qui vous demande de saisir votre nom puis votre prénom et qui affiche ensuite votre nom complet

→ **Algorithme AffichageNomComplet**

**variables** Nom, Prenom, Nom\_Complet: **chaîne de caractères**

**Début**

```
écrire("entrez votre nom")
lire(Nom)
écrire("entrez votre prénom")
lire(Prenom)
Nom_Complet ← Nom & Prenom
écrire("Votre nom complet est : ", Nom_Complet)
```

**Fin**

# Conditions composées

## ■ Les opérateurs logiques combinés

- ➡ 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  $(b = c)$

# Conditions composées

## ■ Les tables de vérités

- ➔ L'évaluation d'une condition composée se fait selon des règles présentées généralement dans ce qu'on appelle des 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>

# Structures conditionnelles

# Instruction conditionnelle

## C'est quoi ?

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

→ On utilisera la forme suivante :

- la condition ne peut être que vraie ou fausse
- si la condition est vraie, ce sont les instructions1 qui seront exécutées
- si la condition est fausse, ce sont les instructions2 qui seront exécutées
- la condition peut être une condition simple ou une condition composée de plusieurs conditions

```
Si condition alors  
instruction ou suite d'instructions1  
Sinon  
instruction ou suite d'instructions2  
Finsi
```

# Instruction conditionnelle

## C'est quoi ?

- La partie **Sinon** n'est pas obligatoire, quand elle n'existe pas et que la condition est fausse, aucun traitement n'est réalisé

```
Si condition alors  
instruction ou suite d'instructions1  
Finsi
```

# Instruction conditionnelle

EXEMPLE

## 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

# Instruction conditionnelle

EXEMPLE

## ■ 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**

# Instruction conditionnelle

EXERCICE

## Exercice 4 Si... Alors

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

# Instruction conditionnelle

EXERCICE  
CORRIGÉ

Exemple Si... Alors

# Instructions conditionnelles imbriquées

## C'est quoi ?

→ Les tests imbriqués / instructions conditionnelles peuvent avoir un degré quelconque d'imbriques

```
Si condition1 alors
    Si condition2 alors
        instructionsA
    Sinon
        instructionsB
    Finsi
Sinon
    Si condition3 alors
        InstructionsC
    Finsi
Finsi
```

# Tests imbriqués

EXEMPLE

## Exemple Si... Alors... Sinon imbriqué

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")

**Finsinon**

**Finsi**

**Fin**

# Tests imbriqués

EXAMPLE

## Exemple Si... Alors... Sinon imbriqué

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

# Test imbriqué

EXERCICE

## Exercice 5

Le prix de photocopies dans une reprographie varie selon le nombre demandé :

- 0,5 euros la copie pour un nombre de copies inférieur à 10,
- 0,4 euros pour un nombre compris entre 10 et 20,
- et 0,3 euros au-delà.

Ecrivez un algorithme qui demande à l'utilisateur le nombre de photocopies effectuées, qui calcule et affiche le prix à payer

# Test imbriqué

EXERCICE  
CORRIGÉ

## Exercice Si... Alors... Sinon

## Exercice 6

1. Déterminer le montant d'un capital  $c$  placé à un taux fixe  $t$  pendant un nombre  $n$  d'années. On suppose que  $c, t, n$  sont lus,

**Exemple de calcul (le taux est de 4% , soit 0,04)**

$$C_n = 10\,000 \times (1+0,04)^5 = 12\,166 \text{ euros, soit un gain de 2\,166 euros.}$$

# Test imbriqué

EXERCICE

## Exercice 7

Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur.  
Ensuite, il l'informe de sa catégorie pour une licence sportive :

- « Baby » de 3 à 6 ans
- « Poussin » de 7 à 8 ans
- « Pupille » de 9 à 10 ans
- "Minime" de 11 à 12 ans
- "Cadet" à partir de 13 ans

# Test imbriqué

EXERCICE

## Exercice 8

Un triangle ( $ABC$ ) est dit isocèle en  $A$  si les côtés  $AB$  et  $AC$  ont même longueur. Il est dit équilatéral si ses trois côtés  $AB$ ,  $BC$  et  $CA$  ont même longueur.

Écrire un programme qui lit les longueurs  $AB$ ,  $BC$  et  $CA$  des côtés d'un triangle et qui affiche le SEUL message correct parmi les messages suivants :

- « Le triangle est équilatéral. » ;
- « Le triangle est isocèle en A mais n'est pas équilatéral. » ;
- « Le triangle est isocèle en B mais n'est pas équilatéral. » ;
- « Le triangle est isocèle en C mais n'est pas équilatéral. » ;
- « Le triangle n'est isocèle ni en A, ni en B, ni en C. ».

*On respectera la contrainte suivante : en plus de n'afficher qu'un seul message, le programme ne doit utiliser AUCUN OPÉRATEUR BOOLÉEN (et, ou, non).*

# Test imbriqué

EXERCICE

## Exercice 9

La figure ci-dessous indique la taille (1, 2 ou 3) d'un vêtement en fonction de la taille d'une personne exprimée en centimètres et de son poids exprimé en kilogrammes. Écrivez un programme qui détermine la taille d'un vêtement en fonction de ces deux critères.

POIDS en kg	TAILLES en cm												
	145	148	151	154	157	160	163	166	169	172	175	178	183
43/47													
48/53				1									
54/59													
60/65									2				
66/71											3		
72/77													

# Structures itératives

## : les boucles

# Instructions itératives : les boucles

## ■ Les types de 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 langage de programmation :

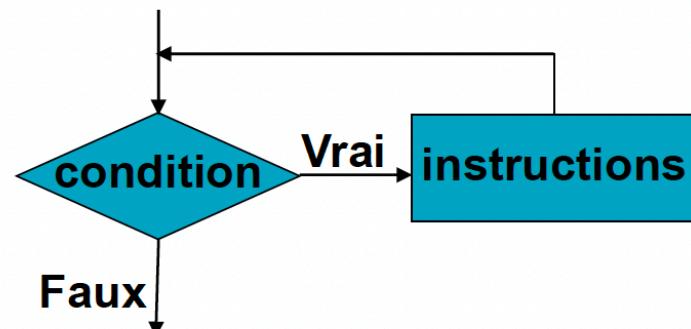
- Les **boucles tant que (while)** : on y répète des instructions tant qu'une certaine condition est réalisée
- Les **boucles jusqu'à (do while)**: on y répète des instructions jusqu'à ce qu'une certaine condition soit réalisée
- Les **boucles pour (for)** ou avec compteur : on y répète des instructions en faisant évoluer un compteur (variable particulière) entre une valeur initiale et une valeur finale

# Les boucles Tant que

## Tant que (while)

- la condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération
- si la condition est vraie, on exécute les instructions (corps de la boucle), puis on teste une nouvelle fois 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

**TantQue** (condition)  
instructions  
**FinTantQue**



# Les boucles Tant que

## Remarques

- 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 :

$i \leftarrow 2$

TantQue( $i > 0$ )

$i \leftarrow i + 1$

FinTantQue

# Boucle Tant que

EXEMPLE

## Tant que

Variable C : caractère

**Debut**

Ecrire (" Entrez une lettre majuscule ")

Lire (C)

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

Ecrire ("Saisie erronée. Recommencez")

Lire (C)

**FinTantQue**

Ecrire ("Saisie valable")

**Fin**

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

# Boucle Tant que

EXEMPLE

## Tant que

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

### version 1 (donne la valeur 14)

Variables som, i : entier

**Debut**

i  $\leftarrow$  0

som  $\leftarrow$  0

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

i  $\leftarrow$  i + 1

som  $\leftarrow$  som+i

**FinTantQue**

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

**Fin**

### version 2 (donne la valeur 15) Attention à l'ordre des instructions

Variables som, i : entier

**Debut**

som  $\leftarrow$  0      L'ordre n'a pas  
i  $\leftarrow$  1      d'impact ici

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

som  $\leftarrow$  som+i

i  $\leftarrow$  i + 1

**FinTantQue**

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

0+1 =1 donc la valeur de i  
n'a pas d'impact non plus  
entre la version 1 et 2

**Fin**

# Boucle Tant que

EXERCICE

## Exercice 10

- ➡ Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

# Test imbriqué

EXERCICE

## Exercice 11

2. Soit un capital  $c$  placé à un taux  $t$ . On veut connaître le nombre d'années nécessaires au doublement de ce capital.

**Exemple de calcul (le taux est de 4% , soit 0,04)**

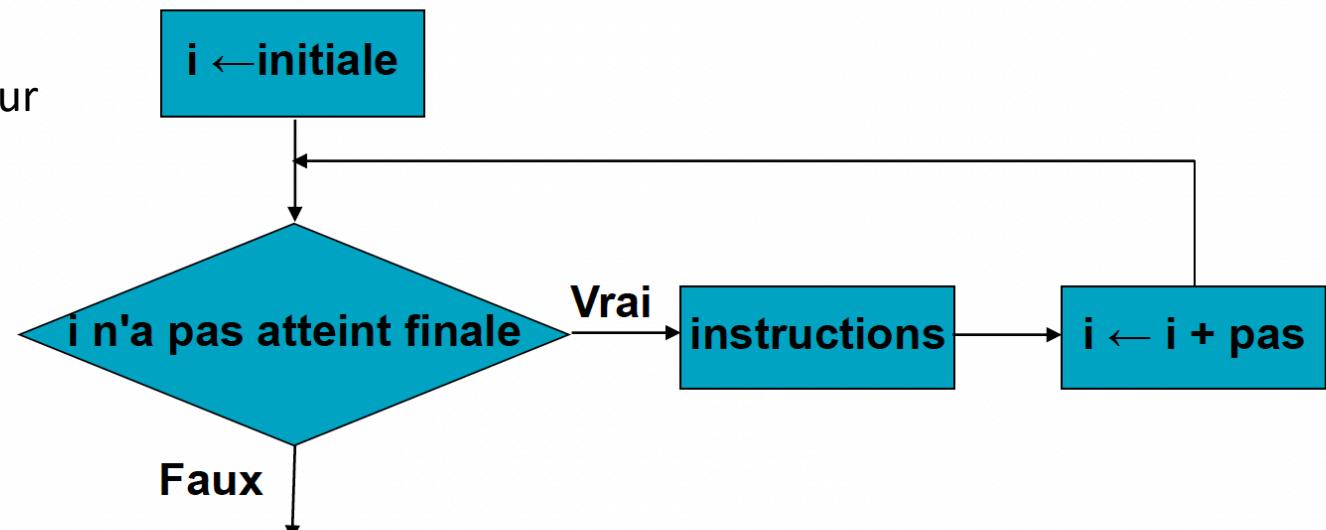
$$C_n = 10\ 000 \times (1+0,04)^5 = 12\ 166 \text{ euros, soit un gain de 2 166 euros.}$$

# Les boucles Pour

## Pour (for)

→ On répète des instructions **en faisant évoluer un compteur** (variable particulière) entre une valeur initiale et une valeur finale

→ le nombre d'itérations dans une boucle Pour est connu avant le début de la boucle



# Les boucles Pour

## ■ Remarques

→ **Compteur** est une variable qui 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 égal 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

**Pour** **compteur** **allant de** **initiale** **à** **finale** **par** **pas** **valeur du pas**

instructions

**FinPour**

# Les boucles Pour

## Pour

- ➔ La valeur initiale est affectée à la variable compteur
- ➔ On compare la valeur du compteur et la valeur 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), les instructions seront exécutées
- ➔ Ensuite, la valeur de compteur est incrémentée de la valeur du pas si pas est positif (ou décrémenté si pas est négatif)
- ➔ On recommence l'étape 2 : La comparaison entre compteur et finale est de nouveau effectuée, et ainsi de suite ...

# Boucle Pour

EXEMPLE

## Pour

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← 1

**Pour i allant de 1 à n**

    puiss← puiss\*x

**FinPour**

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

**Fin**

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

$$X^n = \text{puiss}$$

Exemple :

$$3^2 = 3 \times 3 = 9$$

$$4^5 = 4 \times 4 \times 4 \times 4 \times 4 = 1024$$

# Boucle Pour

EXEMPLE

## Pour

Variables x, puiss: réel

n, i : entier

**Debut**

Ecrire (« Entrez la valeur de x »)

Lire (x)

Ecrire (« Entrez la valeur de n »)

puiss← 1

**Pour i allant de n à 1 par pas -1**

    puiss← puiss\*x

**FinPour**

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

**FinZ**

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)**

# Lien entre Pour et Tant que

## ■ La boucle Pour et un cas particulier de Tant que

→ On utilise la boucle Pour dans le cas où le nombre d'itérations est connu et fixé.

→ **Pour** **compteur** **allant de** **initiale** **à finale** **par pas** **valeur du pas**  
instructions  
**FinPour**

*peut être remplacé par :*

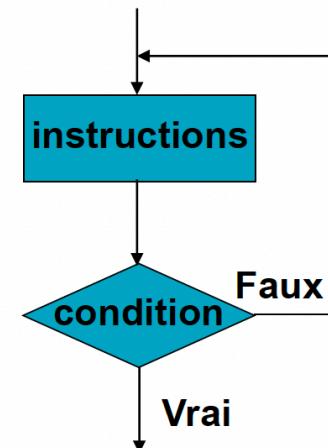
→ **compteur** ← **initiale**  
**TantQue** **compteur** <= **finale**  
instructions  
**compteur** ← **compteur** + **pas**  
**FinTantQue**

# Les boucles Répéter ... Jusqu'à

## Répéter ... Jusqu'à (do.... While)

- La 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 la condition soit vraie (tant qu'elle est fausse)

Répéter  
instructions  
Jusqu'à      condition



# Boucle Répéter ... Jusqu'à

EXEMPLE

## Répéter ... Jusqu'à

Variables som, i : entier

**Debut**

    som← 0  
    i ← 0

**Répéter**

        i ← i + 1  
        som← som+ i

**Jusqu'à ( som> 100)**

    Ecrire (" La valeur cherchée est N= ", SOM )

**Fin**

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

# Les boucles imbriquées

## Boucles imbriquées

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

```
Pour i allant de 1 à 5
    Pour j allant de 1 à i
        Ecrire("O")
    FinPour
    Ecrire ("X")
FinPour
```

```
OX
OOX
OOOX
OOOOX
OOOOOX
```

# Les boucles

## Choix du type de boucle

- ➡ Si on peut **déterminer le nombre d'itérations** avant l'exécution de la boucle, on utilise ***la boucle Pour (FOR)***
- ➡ 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 (while ou do while)***
- ➡ Pour le choix entre ***TantQue (while) et jusqu'à (do while)*** :
  - 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***

## Exercice 12

→ Ecrire l'algorithme permettant d'afficher la table de multiplication par 9.

## Exercice 13

➡ Ecrire un algorithme qui demande successivement 6 nombres à l'utilisateur, et qui lui dit ensuite quel était le plus grand parmi ces 6 nombres.

## Exercice 14

➡ Ecrire un algorithme qui demande un nombre de départ et qui calcule la somme des entiers jusqu'à ce nombre.

Par exemple, si l'on entre 4, le programme doit calculer:  $1 + 2 + 3 + 4 = 10$

## Exercice 15

➡ Ecrire un algorithme qui permet d'afficher les tables de multiplication des nombres de 1 à 10 d'un seul coup

## Exercice 15 bis

→ La ville de Tourcoing a un taux d'accroissement de 0.89 %.

Ecrire un algorithme donnant le nombre d'années nécessaire pour atteindre 120 000 habitants.

On sait qu'en 2015 la ville de Tourcoing comptait 96 809 habitants.

## Exercice 16

➡ Ecrire un programme qui demande à l'utilisateur de saisir 20 notes d'élèves et qui propose le menu suivant :

- Afficher la plus petite note
- Afficher la plus grande note
- Afficher la moyenne des notes

## Exercice 17

➡ Ecrire un programme qui demande à l'utilisateur de saisir des notes d'élèves et qui propose le menu suivant :

- Afficher la plus petite note
- Afficher la plus grande note
- Afficher la moyenne des notes

On arrête la saisie quand l'utilisateur saisie la valeur zéro

## Exercice 17 bis

→ Concevoir un algorithme qui imprime, pour n saisi par l'utilisateur :

1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5...  
1 2 3 4 5 6 ... n

# Les fonctions & procédures

# Fonctions & Procédures

## A quoi ça sert ?

- ➡ 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**, 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

## Rôle d'une fonction

→ Le rôle d'une fonction en programmation est de **retourner 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

**retourne...**

**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 **retourne** sert à retourner la valeur du résultat

# Fonctions

EXEMPLE

## Exemple de fonction

**Fonction** perimetreRectangle (largeur, longueur : entier) : entier

**Variable** perimetre : entier

**Début**

perimetre<—(2\*(largeur+longueur))

**Retourner**(perimetre)

**Fin**

Calcul du périmètre d'un rectangle

# Fonctions

## ■ Appel d'une fonction

- Pour exécuter une fonction, il suffit de faire appel à elle en écrivant son nom suivi des paramètres effectifs.

**Variable** p : entier

**Début**

```
p <- perimetreRectangle(10,15)  
Ecrire("Le périmètre du rectangle est : ", p)
```

**Fin**

**Fonction** perimetreRectangle (largeur, longueur : entier) : entier

**Variable** perimetre : entier

**Début**

```
perimetre<-(2*(largeur+longueur))
```

**Retourner**(perimetre)

**Fin**

# Fonctions

EXERCICE

## Exercice 18

- Définir une fonction qui renvoie le plus grand de deux nombres différents.

Ecrire un programme qui demande deux nombres à l'utilisateur et qui affiche le plus grand des deux nombres en appelant la fonction précédemment créée.

```
Variable p : entier
Début
    p <- perimetreRectangle(10,15)
    Ecrire("Le périmètre du rectangle est : ", p)
Fin
```

```
Fonction perimetreRectangle (largeur, longueur : entier) : entier
Variable perimetre : entier
Début
    perimetre<-(2*(largeur+longueur))
    Retourner(perimetre)
Fin
```

# Procédure

## C'est quoi ?

- 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\_procedure(paramètres et leurs types)**  
Instructions constituant le corps de la procédure  
**FinProcédure**
- Remarque : une procédure peut ne pas avoir de paramètres

# Procédure

## 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.

# Procédure

## ■ Appel d'une procédure

- ➔ Les paramètres servent à échanger des données entre le programme principal (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.

# Procédure

## 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, on va préciser explicitement le mode de transmission dans la déclaration de la procédure

# Fonctions & Procédures

## ■ Portée des variables

→ Une **variable déclarée dans la partie déclaration de l'algorithme principale** est appelée **variable globale**.

Elle est accessible de n'importe où dans l'algorithme, même depuis les fonctions. Elle existe pendant toute la durée de vie du programme.

→ Une **variable déclarée à l'intérieur d'une fonction** est dite **variable locale**.

Elle n'est accessible que dans cette fonction, les autres fonctions n'y ont pas accès. La durée de vie d'une variable locale est limitée à la durée d'exécution de la fonction.

# Les tableaux

# Tableaux

## Pourquoi des tableaux ?

- ➔ 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← 0**

**Si (N1 >10) alors nbre←nbre+1 FinSi**

....

**Si (N30>10) alors nbre←nbre+1 FinSi**

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

# Tableaux

## Pourquoi des 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 :

Variable **tableau** identificateur[dimension] : type

◦ Exemple :

Variable **tableau** notes[30] : réel

- ➔ On peut définir des tableaux de tous types: tableaux d'entiers, de réels, de caractères, de booléens, de chaînes de caractères, ...

# 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
- ➔ Selon les langages, le premier indice du tableau est soit 0, soit 1. Le plus souvent c'est 0.  
Dans ce cas, **notes[i]** désigne l'élément i+1 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
- ➔ Un tableau est inutilisable tant que l'on 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

## Tableaux : exemple

```
Variables i ,nbre : entier
Tableau notes[30] : réel
Début
nbre ← 0
  Pour i allant de 0 à 29
    Si(notes[i]>10) alors
      nbre ← nbre+1
    FinSi
  FinPour
  écrire ("le nombre de notes supérieures à 10 est : ", nbre)
Fin
```

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

# Tableaux

EXAMPLE

## Tableaux : Saisie et affichage

**Procédure** SaisieTab (n : entier par valeur, **tableau** T : réel par référence)

variable i : entier

**Pour** i allant de 0 à n-1

    écrire ("Saisie de l'élément ", i + 1)

    lire (T[i] )

**FinPour**

**Fin Procédure**

**Procédure** AfficheTab(n : entier par valeur, **tableau** T : réel par valeur)

variable i: entier

**Pour** i allant de 0 à n-1

    écrire ("T[",i, "] =", T[i])

**FinPour**

**Fin Procédure**

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

# Tableaux

EXEMPLE

## Tableaux : exemple d'appel

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

**Algorithme Tableaux**

variable p : entier

Tableau A[10]: réel

**Début**

p  $\leftarrow$  10

SaisieTab(p, A)

AfficheTab(p,A)

**Fin**

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

# Tableaux

EXAMPLE

## Tableaux : fonction longueur

**ProcédureSaisieTab( tableau T : réel par référence)**

variable i: entier

**Pour i allant de 0 à longueur(T)-1**

    écrire ("Saisie de l'élément ", i + 1)  
    lire (T[i] )

**FinPour**

**Fin Procédure**

**ProcédureAfficheTab(tableau T : réel par valeur)**

variable i: entier

**Pour i allant de 0 à longueur(T)-1**

    écrire ("T[",i, "] =", T[i])

**FinPour**

**Fin Procédure**

La plupart des langages offrent une fonction **longueur** qui donne la dimension du tableau. Les procédures Saisie et Affiche peuvent être réécrites comme suit :

## Exercice 19

→ Ecrire un algorithme qui déclare et stocke dans un tableau 10 chiffres, puis qui affiche le 9ème élément de ma liste.

# Tableaux

EXERCICE

## Exercice 20

- Écrire un algorithme permettant de saisir 15 notes et de les afficher.

# Tableaux à deux dimensions

## C'est quoi ?

- Les langages de programmation permettent de déclarer des tableaux dans lesquels les valeurs sont repérées par **deux indices**. Ceci est utile par exemple pour représenter des matrices
- En pseudo code, un tableau à deux dimensions se **déclare** ainsi :  
Variable **tableau** identificateur[dimension1] [dimension2] : type
- Exemple : une matrice A de 3 lignes et 4 colonnes dont les éléments sont réels  
Variable **tableau A[3][4]** : réel
- **A[i][j]** permet d'accéder à l'élément de la matrice qui se trouve à l'intersection de la ligne i et de la colonne j

	Etudiant 1	Etudiant 2	Etudiant 3	Etudiant 4
Informatique	12	13	9	10
Comptabilité	12.5	14	12	11
Mathématiques	15	12	10	13

# Lecture d'une matrice

EXAMPLE

## exemple de lecture d'une matrice

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

```
ProcédureSaisieMatrice(n : entier par valeur, m : entier par valeur, tableau A : réel par référence)
Début
    variables i,j: entier
    Pour i allant de 0 à n-1
        écrire ("saisie de la ligne ", i + 1)
        Pour j allant de 0 à m-1
            écrire ("Entrez l'élément de la ligne ", i + 1, " et de la colonne ", j+1)
            lire (A[i][j])
    FinPour
FinPour
Fin Procédure
```

# Affichage d'une matrice

EXEMPLE

## exemple affichage d'une matrice

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

**ProcédureAfficheMatrice(n : entier par valeur, m : entier par valeur, tableauA : réel par valeur)**

**Début**

Variables i,j : entier

**Pour** i allant de 0 à n-1

**Pour** j allant de 0 à m-1

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

**FinPour**

**FinPour**

**Fin Procédure**

# Somme de deux matrices

EXAMPLE

## exemple

Procédure qui calcule la somme de deux matrices :

```
ProcédureSommeMatrices(n, m : entier par valeur, tableauA, B : réel par valeur, tableau C : réel par référence)
Début
    variables i,j : entier
        Pour i allant de 0 à n-1
            Pour j allant de 0 à m-1
                C[i][j] ← A[i][j]+B[i][j]
            FinPour
        FinPour
Fin Procédure
```

# Appel de procédure / matrice

EXAMPLE

## exemple

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 tableauM1[3][4],M2 [3][4],M3 [3][4] : réel

#### 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

# Tableaux

EXERCICE

## Exercice 21

→ Écrire un algorithme permettant la saisie des notes d'une classe de 15 étudiants pour 3 matières.

Algorithme Notes

Constantes N = 15, M = 3 : entier

Variables note : tableau[N, M] de réels i, j : entier

Début

//Remplissage du tableau note

Pour i Allant de 0 à N-1 Faire

    Pour j Allant de 0 à M-1 Faire

        Ecrire("Entrer la note de l'étudiant", i+1, " dans la matière ", j+1)

        Lire(note[i, j])

    FinPour

FinPour

Fin