

# Algorithmique

***CDI-2016***

# Objectif et plan du cours

## ▶ Objectif:

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

## ▶ Plan:

- Généralités (matériel d'un ordinateur, systèmes d'exploitation, langages de programmation, ...)
- Algorithmique (affectation, instructions conditionnelles, instructions itératives, fonctions, procédures, ...)

# Informatique?

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

Applications  
(Word, Excel, Jeux, Maple, etc.)

Langages  
(Java,C/C++, Fortran,etc.)

Système d'exploitation  
(DOS,Windows, Unix, etc.)

Matériel  
(PC, Macintosh, station SUN, etc.)

# Matériel: Principaux éléments d'un PC

## ▶ Unité centrale (le boîtier)

- Processeur ou CPU (*Central Processing Unit*)
- Mémoire centrale
- Disque dur, lecteur disquettes, lecteur CD-ROM
- Cartes spécialisées (cartes vidéo, réseau, ...)
- Interfaces d'entrée-sortie (Ports série/parallèle, ...)

## ▶ Périphériques

- Moniteur (l'écran), clavier, souris
- Modem, imprimante, scanner, ...

# 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, ...)

# Langages informatiques

- ▶ Un langage informatique est un outil permettant de donner des ordres (**instructions**) à la machine
  - A 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

# 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 électrique)
- ▶ Une combinaison de 8 bits= 1 **Octet** →  $2^8 = 256$  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, ?=00111111
- ▶ Les opérations logiques et arithmétiques de base (addition, multiplication, ...) sont effectuées en binaire

# 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

ADD A, 4  
LOAD B              → traducteur      langage machine  
MOV A, OUT

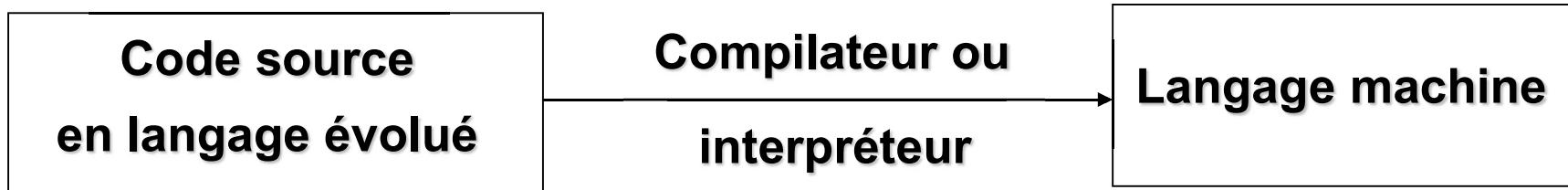
...

- +: 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 évolués

# Langages 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



# Compilateur/interpréteur

- ▶ Compilateur: traduire le programme entier une fois pour toutes



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

- ▶ Interpréteur: traduire au fur et à mesure les instructions du programme à chaque exécution

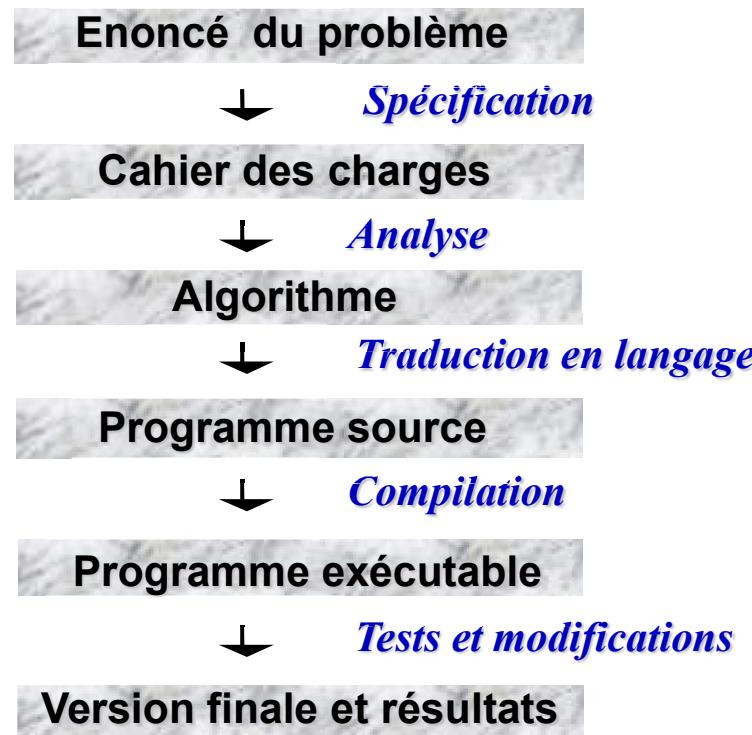


- + exécution instantanée appréciable pour les débutants
- - exécution lente par rapport à la compilation

# Langages de programmation:

- ▶ Deux types de langages:
  - Langages procéduraux
  - Langages orientés objets
- ▶ Exemples de langages:
  - **Fortran, Cobol, Pascal, C, ...**
  - **C++, Java, ...**
- ▶ Choix d'un langage?

# Etapes de réalisation d'un programme



La réalisation de programmes passe par l'écriture d'algorithmes  
⇒ D'où l'intérêt de l'**Algorithmique**

# 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

# 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, elle 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:

- ▶ Un nom doit commencer par une lettre alphabétique  
**exemple valide: A1** **exemple invalide: 1A**
  - ▶ doit être constitué uniquement de lettres, de chiffres et du soulignement \_  
(Eviter les caractères de ponctuation et les espaces) **valides: cdi2016,**  
**cdi\_2016** **invalides: cdi 2016,cdi-2016,cdi;2016**
  - ▶ 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

## Choix des identificateurs (2)

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

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

# Déclaration des variables

- ▶ Rappel: 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**

**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

# L'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:**     $i \leftarrow 1$                    $j \leftarrow i$                    $k \leftarrow i+j$   
 $x \leftarrow 10.3$                    $OK \leftarrow FAUX$                    $ch1 \leftarrow "SMI"$   
 $ch2 \leftarrow ch1$                    $x \leftarrow 4$                    $x \leftarrow j$
- ▶ **non valides:**  $i \leftarrow 10.3$      $OK \leftarrow "SMI"$                    $j \leftarrow x$

# 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 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

# 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 ← 3

B ← 7

A ← B

B ← A + 5

C ← A + B

C ← B – A

**Fin**

# Exercices simples sur l'affectation (2)

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

**Variables A, B : Entier**

**Début**

A ← 1

B ← 2

A ← B

B ← A

**Fin**

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: **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

# 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

# **Exercice (lecture et écriture)**

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

## Tests: instructions conditionnelles (1)

## 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'instructions  
**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

# Tests imbriqués: exercice

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

# Tests imbriqués: corrigé de l'exercice

**Variables** copies : entier

prix : réel

**Début**

Ecrire ("Nombre de photocopies : ")

Lire (copies)

**Si** (copies < 10) **Alors**

    prix ← copies\*0.5

**Sinon Si** (copies < 20) **Alors**

        prix ← copies\*0.4

**Sinon**

        prix ← copies\*0.3

**Finsi**

**Finsi**

Ecrire ("Le prix à payer est : ", prix)

**Fin**

# 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

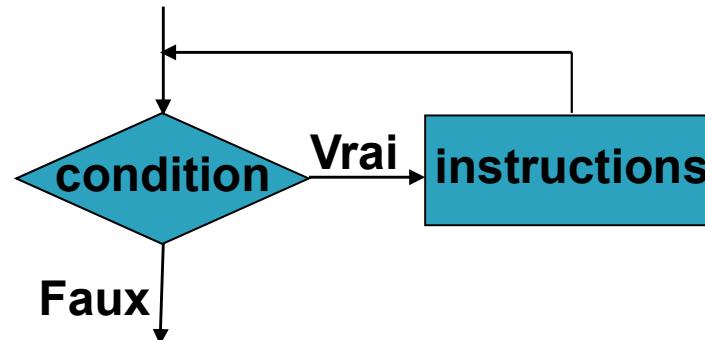
(Dans ce cours, on va s'intéresser essentiellement aux boucles *Tant que* et boucles *Pour* qui sont plus utilisées et qui sont définies en Maple)

# Les boucles Tant que

**TantQue** (condition)

instructions

**FinTantQue**



- ▶ 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 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

- ▶ 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)

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

FinTantQue

# Boucle Tant que : exemple1

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')

Ecrire ("Saisie erronée. Recommencez")

Lire (C)

**FinTantQue**

Ecrire ("Saisie valable")

**Fin**

# Boucle Tant que : exemple2

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  $\leftarrow$  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**

## Boucle Tant que : exemple2 (version2)

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

**Debut**

    som  $\leftarrow$  0

    i  $\leftarrow$  1

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

        som  $\leftarrow$  som + i

        i  $\leftarrow$  i + 1

**FinTantQue**

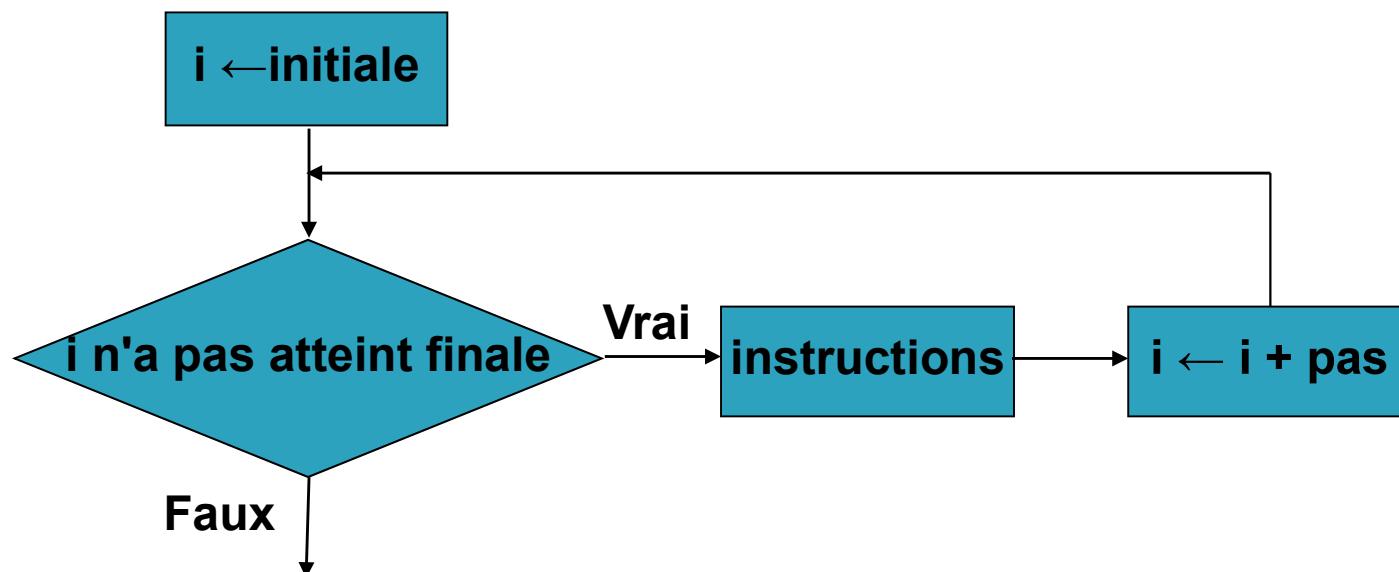
    Ecrire (" La valeur cherchée est N= ", i - 1)

**Fin**

# Les boucles Pour

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

**FinPour**



# Les boucles Pour

- ▶ 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 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

# Déroulement des boucles Pour

- 1) La valeur initiale est affectée à la variable compteur
- 2) 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
    - 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

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**  $i$  allant de 1 à  $n$

    puiss  $\leftarrow$  puiss \*  $x$

**FinPour**

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

**Fin**

# Boucle Pour : exemple1 (version 2)

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 allant de  $n$  à 1 par pas -1**

puiss  $\leftarrow$  puiss \*  $x$

**FinPour**

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

**Fin**

# Boucle Pour : remarque

- ▶ 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 allant de 1 à 5**

$i \leftarrow i - 1$   
**écrire(" i = ", i)**

**Finpour**

# Lien entre Pour et TantQue

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 **allant de initiale à finale par pas** valeur du pas  
instructions

**FinPour**

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

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

**FinTantQue**

# Lien entre Pour et TantQue: exemple

Calcul de  $x^n$  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** ( $i \leq n$ )  
    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 allant de 1 à 5**

**Pour j allant de 1 à i**  
**écrire("O")**

**FinPour**  
**écrire("X")**

**FinPour**

**Exécution**

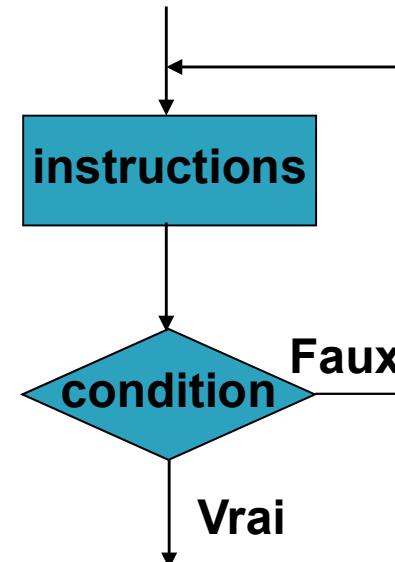
# Les boucles Répéter ... jusqu'à ...

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 vrai (tant qu'elle est fausse)

# Boucle Répéter jusqu'à : exemple

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

**Début**

    som  $\leftarrow$  0

    i  $\leftarrow$  0

**Répéter**

        i  $\leftarrow$  i + 1

        som  $\leftarrow$  som + i

**Jusqu'à ( som > 100)**

Ecrire (" La valeur cherchée est N= ", 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*