**Ir. Olga K. KINYAMUSITU**
**DRCongo**
tpisig1@gmail.com

*« La vie est une suite d'instructions logiques qu'un programmeur se permet de prévoir l'exécution complète et correcte. Il nous faut la façonner comme nous le pouvons, simplifier la complexité des codes pour qu'elle aille comme nous le voulons.»*

# Support du cours d'Algorithmique pour le Premier graduat en Informatique de Gestion

**Appartenant à……**

**Promotion……**

**Année académique 2011-2012**

# Plan du Cours

## *Chap. III : Les fonctions*

*A. Les foncions simples*

*B. l'environnement de données*

*C. Exercices bilan*

## *Chap. IV : Pseudo-codes – Ordinogramme*

*A. Les ordinogrammes*

*B. Les Pseudo-codes*

*C. Exercices d'inter-passation*

## *Chap. V : Du langage algorithmique vers le* langage Visual basique for application

*A. Présentation*

*B. Editeur*

*C. Les variables*

*D. Les procédures et fonctions*

*E. Les tableaux*

*F. Les structures de contrôles et boucles*

*G. Exercices : Annexes-Corrigés des exercices*

## *Conclusion*

## *Bibliographie*

# Introduction

## A. Objectifs du cours

Joseph Campbell a dit que l*es ordinateurs sont comme les dieux de l'Ancien Testament : avec beaucoup de règles, et sans pitié ;* c'est ainsi que le présent cours essayera, dans toute son extension, de pénétrer les pensées de ces dieux en comprenant le langage algorithmique sous les objectifs suivants :

- L'étudiant sera capable, à la fin de ce cours, de connaître tous les concepts relatifs à la logique algorithmique ;
- Il sera capable de distinguer un pseudo-code d'un ordinogramme pour ainsi structurer, sans trop de peine un programme ;
- Il pourra passer d'un pseudo-code à un code de programme sous n'importe quel langage, et surtout ici le Visual basique ;
- L'étudiant sera aussi capable de bien concevoir un algorithme et un programme
- il aura une initiation basique à la programmation, tout en insistant sur les notions de conception et de réflexion ;

En bref, le cours d'algorithme prépare l'étudiant en Informatique de Gestion, au raisonnement informatique pour obtenir le qualificatif d'analyste programmeur de tout système d'information.

## B. Généralités sur les Algorithmes

- ✓ Définition de quelques concepts

Le mot **Algorithmique** est un terme d'origine arabe, comme algèbre, amiral ou zénith. Ce n'est pas une excuse pour massacrer son orthographe, ou sa prononciation. C'est aussi un ensemble de règles opératoires rigoureuses, ordonnant à un processeur d'exécuter dans un

ordre déterminé un nombre fini d'opérations élémentaires ; il oblige à une programmation structurée.

Ce terme a été simplifié par bon nombre de scientifiques sous le concept d'algorithme, qui se définit comme une suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné. C'est ainsi que Si l'algorithme est juste, le résultat est le résultat voulu, et Si l'algorithme est faux, le résultat est, disons, aléatoire.

La maîtrise de l'algorithmique requiert deux qualités, très complémentaires d'ailleurs :

- il faut avoir une certaine **intuition**, car aucune recette ne permet de savoir a priori quelles instructions permettront d'obtenir le résultat voulu. C'est là, qu'intervient la forme *d'intelligence* requise pour l'algorithmique. Mais alors, c'est certain, il y a des gens qui possèdent au départ davantage cette intuition que les autres. Cependant, il est important d'insister sur ce point, les réflexes, cela s'acquiert. Et ce qu'on appelle l'intuition n'est finalement que de l'expérience tellement répétée que le raisonnement, au départ laborieux, finit par devenir *spontané*.
- il faut être **méthodique** et **rigoureux** : chaque fois qu'on écrit une série d'instructions qu'on croit justes, il faut systématiquement se mettre mentalement à la place de la machine qui va les exécuter, armé d'un papier et d'un crayon, afin de vérifier si le résultat obtenu est bien celui que l'on voulait. Cette opération ne requiert pas la moindre portion d'intelligence. Mais elle reste néanmoins indispensable, si l'on ne veut pas écrire à l'aveuglette.

N.B : Naturellement, cet apprentissage est long, et demande des heures de travail patient. Aussi, dans un premier temps, évitez de sauter les étapes (la vérification méthodique, pas à pas, de chacun de vos algorithmes représente plus de la moitié du travail à accomplir.)

Pour fonctionner, **un algorithme doit contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter**. C'est d'ailleurs l'un des points délicats pour les rédacteurs de modes d'emploi : les références culturelles, ou lexicales, des utilisateurs, étant variables, un même mode d'emploi peut être très clair pour certains et parfaitement incompréhensible pour d'autres.

En informatique, heureusement, il n'y a pas ce problème : les choses auxquelles on doit donner des instructions sont les ordinateurs, et ceux-ci ont le bon goût d'être tous strictement aussi « idiots » les uns que les autres et ne sont fondamentalement capables de comprendre que quatre catégories d'ordres (en programmation, on n'emploiera pas le terme d'ordre, mais plutôt celui d'**instructions**). Ces **quatre** familles d'instructions sont :

- l'affectation de variables
- la lecture / écriture
- les tests
- les boucles

Un algorithme informatique se ramène donc, au bout du compte, à la combinaison de ces quatre petites briques de base. Il peut y en avoir quelques unes, quelques dizaines, et jusqu'à plusieurs centaines de milliers dans certains programmes de gestion. Rassurez-vous, dans le cadre de ce cours, nous n'irons pas jusque là (cependant, la taille d'un algorithme ne conditionne pas en soi sa complexité : de longs algorithmes peuvent être finalement assez simples, et de petits très compliqués).

✓ Programmation ou Algorithmique

Il est important de relever le petite nuance de différence entre les termes programme et algorithme, bien que pas totalement différents.

Certains auteurs affirment que l'algorithme est une méthode pour résoudre un problème, alors que le programme est le codage lisible par l'ordinateur de cette méthode.

Avant d'écrire un programme, il est nécessaire d'avoir un algorithme. Il s'agit donc de fournir la solution à un problème, la première étape consiste donc à analyser le problème, c'est-à-dire en cerner les limites et le mettre en forme dans un langage descriptif. On parle généralement d'analyse pour décrire le processus par lequel le problème est formalisé. Le langage de description utilisé pour écrire le résultat de l'analyse est appelé ***algorithme***.

Un programme est considéré comme une suite d'instructions permettant de réaliser une ou plusieurs tâches, de résoudre un problème, de manipuler les données. C'est l'expression d'un algorithme dans un langage donné pour une machine donnée ; tandis qu'un algorithme est une séquence d'opérations visant à la résolution d'un problème en un temps fini (mentionnez la condition d'arrêt).

L'étape suivante consiste à traduire l'algorithme dans un langage de programmation spécifique, il s'agit de la phase de programmation.

C'est ainsi il est plus que nécessaire d'apprendre l'algorithmique pour apprendre à programmer car l'algorithmique exprime les instructions résolvant un problème donné **indépendamment des particularités de tel ou tel langage**. A titre illustratif, si un programme était une dissertation, l'algorithmique serait le plan, une fois mis de côté la rédaction et l'orthographe. Or, vous savez qu'il vaut mieux faire d'abord le plan et rédiger ensuite que l'inverse.

Apprendre l'algorithmique, c'est apprendre à manier la **structure logique** d'un programme informatique. Cette dimension est présente quelle que soit le langage de programmation ; mais lorsqu'on

programme dans un langage (en C, en C++, en Visual Basic, etc.) on doit en plus se colleter les problèmes de syntaxe, ou de types d'instructions, propres à ce langage. Apprendre l'algorithmique de manière séparée, c'est donc sérier les difficultés pour mieux les vaincre.

✓ Ecriture d'un Algorithmique

L'algorithmique possède deux grandes écritures, bien que auparavant, plusieurs types de notations ont représenté des algorithmes.

Il y a eu notamment une représentation graphique, avec des carrés, des losanges, etc. qu'on appelait des **organigrammes**. Aujourd'hui, cette représentation est quasiment abandonnée, pour deux raisons :

- D'abord, parce que dès que l'algorithme commence à grossir un peu, ce n'est plus pratique du tout.
- Ensuite parce que cette représentation favorise le glissement vers un certain type de programmation, dite non structurée, que l'on tente au contraire d'éviter.

C'est la raison pour laquelle les deux types de représentation d'un algorithme sont :

- une série de conventions appelées « **pseudo-code** », qui ressemble à un langage de programmation authentique dont on aurait évacué la plupart des problèmes de syntaxe. Ce pseudo-code est susceptible de varier légèrement d'un livre (ou d'un enseignant) à un autre. C'est bien normal : le pseudo-code, encore une fois, est purement conventionnel ; aucune machine n'est censée le reconnaître.
- Une autre appelée **ordinogramme** ; à ne pas confondre avec l'organigramme que nous avions déjà définit dans les pages précédentes. L'ordinogramme ; lui est la représentation graphique

d l'algorithme. Nous aurons à développer son formalisme dans le chapitre deux.

✓ Structure d'un algorithme

Le langage de description d'algorithme utilise un ensemble de mots clés et de structures permettant de décrire de manière complète et claire, l'ensemble des opérations à exécuter sur des données pour obtenir des résultats. L'avantage d'un tel langage est de pouvoir être facilement transcrit dans un langage de programmation structurée.

La structure d'un algorithme comprend :

- L'en-tête : permet tout simplement d'identifier un algorithme
- Les déclarations : sont des listes exhaustives des objets, grandeurs utilisées et manipulées dans le corps de l'algorithme. Cette liste est placée en début d'algorithme
- Le corps : dans cette partie, sont placées les tâches (instructions, opérations) à exécuter.
- Les commentaires : pour permettre une interprétation aisée de l'algorithme, leur utilisation est vivement conseillée.

Tous les mots clés sont soulignés et écrits en **minuscule.**
Une marque de terminaison (;) est utilisée entre chaque action.

Ainsi, il sera facile de continuer puisque toutes les notions relatives aux Algorithmes ont été abordées et surtout il sied de retenir que cet apprentissage est long et demande des heures de travail patient. Aussi dans un premier temps il faut éviter de sauter des étapes ; la vérification de chacun de vos algorithmes doit être considérée comme la moitié du travail à accomplir

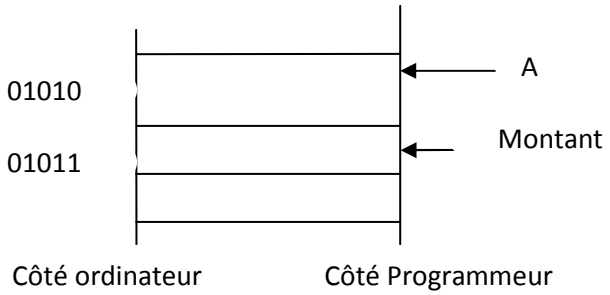
# Chap. I : Les Variables

La plupart de langages sont basés sur la même technique fondamentale, celle de manipulation de valeurs contenues dans des variables.

L'écriture d'un programme est une opération complexe qui requiert de nombreuses étapes. Le plus important est de comprendre l'objectif final et de le respecter.

## A. Structure d'un algorithme

En programmation, une variable est donc un nom qui sert à repérer un emplacement donné de la mémoire centrale. Elle vous permet, en effet de manipuler de valeur sans avoir à vous préoccuper de l'emplacement qu'elles occuperont effectivement en mémoire.




Le programmeur ne se contente que des noms A et Montant, il ne se préoccupe pas des adresses qui leurs seront attribués en mémoire.

La première chose à faire avant de pouvoir utiliser une variable est de créer la boîte et de lui coller une étiquette .Ceci se fait tout au début de l'algorithme avant même les instructions proprement dites. C'est ce qu'on appelle la ***déclaration des***

***variables.*** Le nom de variable (étiquette de la boîte) obéit à des impératifs changeant selon les langages.

En pseudo-code, une déclaration de variable aura cette forme :

**Variable : <Nom_variable> : <Type_de_la_variable> ;**

Ex : Variable : rst : entier ;

**Quelques limites dans la déclaration des variables**

Une règle absolue est qu'un nom de variable peut comporter des lettres et des chiffres, mais qu'il exclut la plupart des signes de ponctuation, en particulier les espaces.

Mais, Dans beaucoup de langage de programmation le caractère "_" est utilisé dans la déclaration des variables, Même au début. Le nom d'une variable commence aussi impérativement par une lettre. Le nombre maximal de signes pour un nom de variable dépend du langage utilisé.

**Structure proprement dite**

Un algorithme a comme propriété de fournir toujours le même résultat. La structure d'un algorithme comprend 2 parties :

- la première ligne indique le nom de l'algorithme
- la deuxième partie es le traitement et est situé entre Debut et Fin, contient le bloc d'instructions.

Un bloc d'instructions est une partie du traitement d'un algorithme constitué d'opérations élémentaires situées entre Debut et Fin ou entre accolades.

Algorithme Nom_de_l'algorithme //partie en-tête

Debut

//partie traitement

Bloc d'instructions ;

Fin

**//** : Commentaires (explications textuelles inscrites dans l'algorithme par le programmeur et ne seront jamais exécutés car sont invisibles lors de l'exécution de l'algorithme.

N.B : Chaque ligne comporte une seule instruction. L'exécution de l'algorithme correspond à la réalisation de toutes les instructions, ligne après ligne, de la première à la dernière selon l'ordre.

Exemple : Algorithme qui permet d' afficher BONJOUR ISIG.

Algorithme Affiche

Début

ecrire (‘’ BONJOUR ISIG’’) ;

End

## B. Les données

- *Déclaration et utilisation de variables*

Les valeurs, pour être manipulées, sont stockées dans de variables. Une variable se caractérise par :

- un nom unique qui la désigne
- un type de définition unique
- une valeur attribuée et modifiée au cours de l'exécution de l'algorithme.

Dans tous les cas, les variables utilisées au cours de l'exécution de l'algorithme sont déclarées immédiatement après le nom de l'algorithme

Il suffit d'indiquer le nom de la variable, suivi de son type et séparé de deux points.

**Algorithme Nom_de_l'algorithme //partie** //partie de déclaration de variables

**variable : rst: entier;**

**Début** //partie traitement

**Bloc d'instructions**

**Fin**

- *Nom, type et valeur*

Le nom d'une variable permet d'identifier de manière unique la variable au cours de l'exécution de l'algorithme. On l'appelle aussi identifiant de la variable. Le nom de la variable ne comporte pas d'espace et s'il est composé de plusieurs mots, il faut faire commencer chacun d'eux par une majuscule(Ex : SalaireDeBase).

Le type (domaine de définition) d'une variable indique l'ensemble de valeurs qu'une variable peut prendre. Les variables peuvent appartenir à plusieurs domaines (entier, caractère, réel, booléen) et chacun étant associé à des opérations spécifiques.

Dans tous les langages, toutes les variables d'un même type occupent en mémoire un nombre déterminé de bits. En toute rigueur, il fait dire qu'une exception a leur pour le type chaine qui existe dans certains langages.

Le type d'une variable définit :

- la nature des informations qui seront représentées dans la variable (numériques, caractères)
- le codage utilisé
- les limitations concernant les valeurs qui peuvent être représentées
  les opérations réalisables avec les variables correspondantes.

La valeur de la variable est la seule caractéristique qui soit modifiée au cours de l'algorithme. Au début de l'algorithme, toutes les variables ont des valeurs inconnues. Les variables changent de valeurs grâce à l'opération d'affectation.

L'affectation est une opération qui fixe une nouvelle valeur à une variable. le symbole d'affectation est ←

Exemple : Algorithme d'obtention du double de 7.

Algorithme Produit

Variable : nbre1 : entier;

variable : resultat : entier;

Début

nbre1 ← 7;

resultat ← nbr1 * 2;

ecrire ("resultat");

Fin

Pour écrire un algorithme, il faut commencer par définir l'ensemble de variables nécessaires à son traitement. Il peut sembler simple de définir les données et le résultat du problème, mais la moindre erreur dans la détermination de bonnes variables à utiliser au cours d'un algorithme est lourde de conséquence. Lorsque le problème a été décomposé en une suite d'opérations, il faut toutes les résoudre.

Dans le langage algorithmique, il existe de variables, de type entier, caractère et booléens.

- Le type réel et entier, en anglais ***Integer*** est utilisé dans un algorithme que pour les variables ayant comme domaines usuels ceux fournis par les mathématiques. Les opérations utilisables sur les éléments de ces domaines sont toutes les opérations mathématiques (+, -, *, /) et comme opérateurs, ceux de comparaison (<, >, =, >=, <=, <>).

- Le type caractère, en anglais ***String***, ne s'intéresse qu'au domaine constitué de caractères alphabétiques, alphanumériques et de ponctuation. Les seules opérations élémentaires pour les éléments de type caractère sont les opérations de comparaison.

- Le type booléen en anglais ***Boolean*** a comme domaine l'ensemble formé de deux seules valeurs (vrai, faux). Les opérateurs admissibles sur les éléments sont réalisées à l'aide de tous les connecteurs logiques, notés : ET (and), OU (or), NON (not).

| Opérateur ET | Faux | Vrai |
|---|---|---|
| Faux | Faux | Faux |
| Vrai | Faux | Vrai |

| Opérateur Ou | Faux | Vrai |
|---|---|---|
| Faux | Faux | Vrai |
| Vrai | Vrai | Vrai |

N.B : Lors d'une affectation, la valeur de la partie droite doit obligatoirement être du type de la variable dont la valeur est modifiée.

### C. Fonctions d'entrée-sortie

Les algorithmes ont pour vocation de nous faire réfléchir sur papier. Il est nécessaire de simuler le déroulement de notre algorithme.

Les programmes utilisent fréquemment des instructions permettant l'affichage à l'écran et la saisie de valeurs au clavier par l'utilisateur.

- ***La fonction Ecrire***

Cette instruction a pour rôle de nous fournir des résultats sous une forme directement compréhensible. Plus précisément, elle « Ecrit» sur un périphérique les valeurs d'une ou de plusieurs variables.

Une telle instruction ne se contente pas de transmettre à un périphérique le simple binaire de variables concernées (nous aurions certainement quelque peine à l'interpréter). Elle doit également transformer ce contenu en un ou plusieurs caractères compréhensibles par l'utilisateur.

L'instruction d'affichage à l'écran (périphérique de sortie) d'une expression est :

***ecrire (''expression'') ;***

Exemple : Algorithme qui affiche Premier message

Algorithme Message

Debut

ecrire (''Premier message'') ;

Fin

- ***La fonction Lire***

Cette instruction permet au programme de nous communiquer des résultats. de manière analogue, l'instruction de lecture va nous permettre de fournir de valeurs à notre programme. Plus précisément, cette instruction va « chercher» une valeur sur un périphérique et attribue à une variable.

Dans la plupart de langages, il est possible de choisir le périphérique sur lequel on souhaite lire. L'exécution de cette instruction consiste à demander à l'utilisateur de saisir une valeur sur le périphérique d'entrée et de modifier la valeur de la variable passée entre parenthèses.

L'instruction d'affichage à l'écran (périphérique de sortie) d'une expression est :

***lire (expression) ;***

Avant l'exécution de cette instruction, la variable de la liste avait ou n'avait pas de valeur ; après elle a la valeur lue au clavier (le périphérique d'entrée).

Le langage algorithmique nous permet dès à présent de résoudre de petits problèmes. Nous nous baserons sur ces fonctions dans la suite de notre cours.

### D. Les types objets

Pour un cours d'algorithmique en Première année de graduat, cette notion d'objets semblera difficile à comprendre. Nous l'insérons à titre d'information et cela pour permettre aux étudiants de se préparer à la programmation orienté objet.

Nous avons introduit les types primitifs *entier, caractère, booléen* avec les opérateurs usuels associés. De la même manière, le type objet permet la manipulation de données : le type *chaine* et le type *date*.

Une chaine de caractère est composée de caractères alphanumériques formant un mot ou une phrase. Il est impossible de manipuler les chaines de caractère avec les opérateurs usuelles définies pour les réels ou les entiers : la classe chaine nous fournit donc des opérations définies.

Une date est un type permettant de gérer les dates désignées par le jour, le mois et l'année. Avec la classe *Date*, il convient de fournir l'ensemble des opérations capables de gérer des dates.

### E. Les schémas mémoires

Il est primordial de connaitre l'état des variables en cours d'exécution d'un algorithmique grâce à un schéma mémoire. Un schéma mémoire délimite trois parties distinctes :

- Le nom de l'algorithme ;
- La partie des variables où toutes les variables définies seront représentées par :
    - une valeur pour les variables de type primitif ;
    - une flèche pour les variables de type chaine ou Date.
- La partie des instances où chaque case aura été créée par l'utilisation de l'opérateur ***new.*** Il ya autant de case à représenter qu'il ya de ***new*** dans l'algorithme.

N.B : Il faut éviter d'oublier de représenter une variable, de donner une mauvais valeur à une variable, de représenter les chaines ou les dates sans une case associée.

Le schéma mémoire d'un algorithme représente l'ensemble de variables et de leurs valeurs à une étape précise de son déroulement.

### F. le type tableau

Le type tableau permet de stocker des valeurs de même type grâce à une seule variable. La notion de tableau consiste à :

- Attribuer un seul nom à l'ensemble de valeurs se trouvant dans une variable ;

- Repérer chaque valeur par ce nom suivi de crochets d'un certain numéro de valeurs.

Donc, de façon plus simple, un tableau structure un ensemble de valeurs de même type accessibles par leur position.

- *Dimension, type et indice*

Le nombre maximal d'éléments du tableau, qui est précisé à la définition s'appelle ***dimension***. Le type de ses éléments s'appelle ***type*** du tableau, alors que pour accéder aux éléments d'un tableau, un ***indice*** indique le rang de cet élément. Un tableau est utilisé en temps : sa déclaration et l'instruction dans le corps de l'algorithme.

*Syntaxe de déclaration du tableau* :

**variable : nom_Variable : tableau [] de type_variable** ;

L'instruction dans le corps du programme réserve de la place en mémoire et est construite avec un operateur ***new*** agissant dans l'environnement d'exécution du programme :

**nom_tableau ⟵new domaine [dimension]**;

Exemple : L'exemple suivant représente le tableau d'entiers tab (d'indicateur tab, de dimension 10 et dont le type). Les indices qui permettent d'y accéder vont de 0 à 9.

| tab = | 21 | -7 | 12 | 23 | 40 | -5 | 23 | 4 | 27 | 36 |
|---|---|---|---|---|---|---|---|---|---|---|
| Indices | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |

Notons que le premier indice des éléments d'un tableau vaut toujours **0** et le dernier vaut la dimension du **tableau – 1**. A l'aide d'une seule variable de type tableau, il est possible de

manipuler plusieurs valeurs différentes. Par exemple tab [0] est vu comme une variable indépendante ayant comme indice 21, tab [5] est une autre variable de valeur – 5…

- *Utilisation d'un tableau*

Pour comprendre comment un tableau fonction, il faut, à cette étape, faire allusion aux tableaux Excel où une cellule s'obtient par le produit cartésien de colonnes (lettes) et de lignes (chiffes).

➢ *Tableau à une dimension*

La manipulation des éléments de ce type de tableau se fait de telle manière qu'une seule donnée stocke une ou plusieurs valeurs de même type.

Exemple :

Algorithme une-dimension

Variable : note : tableau [ ] d'entiers;

Debut

note ⟵ new entier [4 ] ;

note [0 ] ⟵ 12;

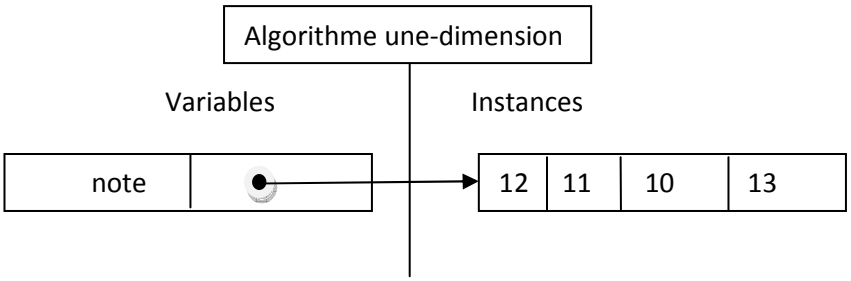
note [1 ] ⟵ 11;

note [2] ⟵10;

note [3 ] ⟵13;

Fin

Voici l'état mémoire à la fin de l'exécution de cet algorithme :




- *Tableau à deux dimensions*

Si nous désirons écrire un programme qui travaille avec un damier de 3 cases sur 3 contenant des entiers, nous devons introduire une instance *rst* sous forme d'un tableau de 3 cases sur 3.

Algorithme deux-dimension

Variable : rst : tableau [ ] [ ] d'entiers;

Debut

note new ⟵ entier [4] ;

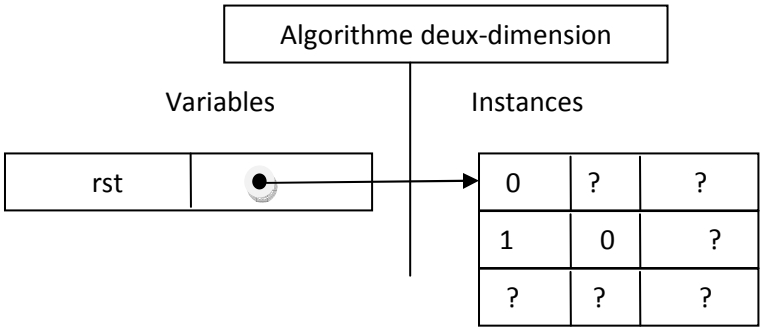
note [0] [0] ⟵ 0; //première ligne et première colonne

note [1] [1] ⟵ 0;

note [0] [1] ⟵ 1;

Fin

Voici l'état mémoire à la fin de l'exécution de cet algorithme :




## G. Exercices pratiques : *A faire avec les étudiants*

1. Ecrire un programme qui permet de faire la somme entre deux nombres entrés au clavier
2. Faire le programme imprimant une facture.
3. Faire un programme qui calcule le carré entre deux nombres entrés au clavier
4. Ecrire un programme qui affiche les lettres votre nom dans un tableau.

# *Chap. II : Les structures de contrôle*

Pour construire un algorithme au déroulement non linéaire, nous avons besoin de deux instructions extrêmement importantes : l'instruction conditionnelle et la boucle.

L'instruction conditionnelle permet d'exécuter ou non un bloc d'instructions, alors que la boucle permet de revenir en arrière dans l'algorithme, pour réitérer un nombre de fois précis l'exécution d'un bloc.

Ces deux instructions reposent sur l'évaluation, par l'algorithme, d'un variable de type booléenne (vrai ou faux), qui conditionne la suite de son déroulement.

## A. **L'instruction conditionnelle**

Un algorithme est constitué d'une suite d'instructions qui s'exécutent les unes après les autres de la première à la dernière ligne.

L'instruction conditionnelle nous autorise à concevoir un algorithme qui n'exécutera pas certains blocs instructions. Elle est définie comme étant celle qui détermine si le bloc d'instructions suivant est exécuté ou non. La condition est une expression booléenne dont la valeur détermine le bloc d'instructions exécutées.

- *Syntaxe de l'instruction conditionnelle* :

**Si (condition) alors**

{

**Bloc d'instructions n°1 // exécutée si condition est vraie**

}

**Sinon**

{

**Bloc d'instructions n°2 // exécutée si condition est Fausse**

}

Le bloc d'instruction exécuté dépend de la valeur booléenne. Si la condition vaut vrai, seul le bloc numéro 1 est exécuté, le bloc numéro 2 ne sera pas exécuté. Si la condition vaut faux, seul le bloc d'instructions numéro 2 est exécuté.

Exemple : écrivons un algorithme qui permet de lire 2 entiers et affiche le plus grand d'entre eux.

**Résolution** : l'analyse de cet énoncé fait ressortir deux données nécessaires (les deux entiers à lire) et un résultat (entier lui aussi).

Algorithme grand

Variables : nbre1, nbre2, max : entier ;

Debut

Ecrire(''valeur du premier nombre :'') ;

Lire(nbre1) ;

Ecrire(''valeur du deuxième nombre :'') ;

Lire(nbre2) ;

Si (nbr1>nbr2) alors

{

max ⟵ nbr1 ;

}

Sinon

{

max ⟵nbre2 ;

}

Ecrire (''Le plus grand nombre est : '', max) ; //idem lire (max) ;

Fin

- *La conditionnelle simple*

Une version plus simple est utilisée si l'alternative n'a pas lieu. La syntaxe de cette instruction est alors :

**Si (condition) alors**

**{**

**Bloc d'instructions**

**}**

Écrivons un algorithme qui lit un entier et affiche sa valeur positive.

```
Algorithme positif
Variables : y : entier ;
Variables : x : texte ;
Debut
    ecrire(''valeur du nombre :'') ;
    lire(x) ;
      y  ←x ;
Si (x < 0) alors
{
      y  ←  -1 *x ;
}
Écrire (''La valeur positive est : '', y) ;
Fin
```

Le même algorithme peut se passer de la variable y contenant le résultat :

```
Algorithme positif
Variables : x : entier ;
Debut
    Ecrire(''valeur du nombre :'' ,x) ;
Si (x < 0) alors
```

{

x ⟵ -1 *x ;

}

Écrire (‘’La valeur positive est : ‘’, x) ;

Fin

On peut supprimer l’écriture de l’instruction contenant les accolades de délimitation de bloc, lorsqu’il n’y a pas d’ambigüité (si le bloc n se compose que d’une seule instruction).

Algorithme maximum

Variables : x, y, max : entier ;

Debut

ecrire(‘’valeur du nombre :’’ ,x) ;

ecrire (‘’la valeur de l’autre nombre’’, y) ;

Si (x > y) alors

max ⟵ x ;

Sinon

max ⟵ x ;

Écrire (‘’La valeur est : ‘’, max) ;

Fin

Remarquons bien que l’instruction *si alors sinon* est une seule instruction, composée d’une condition et de deux blocs d’instructions.

Dans un premier temps, nous allons utiliser les accolades (vu notre niveau en algorithmique) même si le bloc d'instructions est réduit à une seule instruction.

- *La présentation*

Les décalages dans l'écriture d'un algorithme (ou d'un programme) sont nécessaires à sa bonne lisibilité. Savoir présenter un algorithmique, c'est montrer qu'on a compris son exécution.

La règle est assez simple : dès qu'un nouveau bloc d'instructions commence par un *Debut*, toutes les lignes suivantes sont décalées d'une tabulation. Dès qu'un bloc se termine par un Fin, toutes les lignes suivantes sont décalées d'une tabulation vers la gauche.

### B. **Conditionnelles imbriquées**

- *Usage*

Partons d'un exemple, pour expliquer cette notion de conditionnelles imbriquée. Il est possible d'imbriquer de blocs de programmes les unes dans les autres.

Essayons de résoudre le problème consistant à faire lire à l'utilisateur note et à afficher le commentaire associé à la note :

- Note de 0 à 8 on affiche :''insuffisant''
- Note de 8 à 12 on affiche :''moyen''
- Note de 12 à 16 on affiche :''bien''
- Note de 16 à 20 on affiche :''très bien''

La première instruction utilise la conditionnelle classique :

```
Algorithme note
Variable : note : entier ;
Debut
    lire (note) ;
Si (note < = 8) alors
    ecrire(''insuffisant'') ;
Si (note >8) ET  (note< = 12) alors
    ecrire('' moyen'') ;
Si (note >12) ET  (note<= 16) alors
    ecrire(''bien'') ;
Si (note >16) alors
    ecrire('' très bien'') ;
Fin
```

A chaque note saisie, 4 tests sont réalisés. Nous pouvons écrire cet algorithme de façon plus élégante :

```
Algorithme note_vraie
Variable : note : entier ;
Debut
    Lire (note) ;
Si (note <=8) alors
```

```
        ecrire(''insuffisant'') ;
Sinon si  (note<= 12) alors
            ecrire('' moyen'') ;
        Sinon si (note<= 16) alors
                    ecrire(''bien'') ;
                    Sinon (note >16)
                            ecrire('' très bien'') ;
Fin
```

En effet, dans tout le bloc *sinon* du premier test, nous sommes certains que la note (la valeur de la variable note) est strictement supérieure à 8, donc il est inutile de refaire ce test.

Dans ce cas, avec les conditionnelles imbriquées, il est possible de ne pas respecter la tabulation.

N.B : Il a été remarqué que chez les débutants d'oublier les instructions *sinon* intermédiaires.

Analysons cet algorithme (faux) pour ne pas commettre cette erreur :

```
Algorithme erreur
variable : notes : entier ;
Debut
        Lire (note)
        Si (note < =8) alors
```

ecrire (‘’insuffisant’’) ;

Si (note < =12) alors

ecrire(‘’moyen’’) ;

Si (note < =16) alors

ecrire(‘’bien’’) ;

Sinon

ecrire (‘’très bien’’) ;

Fin

Remarquer que pour une note inférieure ou égale à 8, l’algorithme précédent écrira effectivement ‘’insuffisant’’, mais aussi ‘’moyen’’ et ‘’bien’’.

En effet, chacune des conditions est dans ce cas testée indépendamment des instructions précédentes.

- *Présentation*

Il important d’écrire un algorithme aussi lisible et clair que possible :

Si (condition) alors

{

Bloc d’instructions 1 ;

}

Sinon

{

```
Si (condition) alors
{
Bloc d'instructions 2;
}
Sinon
{
Bloc d'instructions 3 ;
}
}
```

Dans un bloc délimité par *Debut* et *Fin*, toutes les lignes s'exécutent les unes après les autres : elles sont toutes décalées.

C. **Instructions de répétition**

Une répétition sert à répéter un ensemble d'instructions jusqu'à ce qu'une certaine condition soit réalisée.

- *La boucle tant_que*

L'instruction de répétition, dite boucle, permet d'exécuter plusieurs fois consécutives un même bloc d'instructions. La répétition s'effectue tant que la valeur de l'expression booléenne est égale à **vraie**.

Il faut savoir que l'instruction de répétion du déroulement d'un bloc d'instructions la plus classique est la boucle ***tant_que.*** Sa syntaxe est particulièrement simple ; on veut contrôler la répétition de l'exécution d'un bloc.

L'instruction précise une condition de répétition qui conduit la poursuite ou l'arrêt de l'exécution du bloc d'instructions.

Syntaxe :

tant_que (condition_de_poursuite) faire

{

Bloc d'instructions ;

}

**On utilisera une boucle tant_que quand l'algorithme doit effectuer plusieurs fois le même traitement, lorsqu'il doit répéter un ensemble d'instructions.** C'est la seule instruction qui permette en quelque sorte de revenir en arrière dans l'algorithme pour exécuter une même série d'instructions.

Exemple : Affichons à l'écran les entiers de 1 à 4.

Algorithme affichage_entier

Variables : compteur : entier ;

Debut

compteur ⟵ 1 ; //initialisation

tant_que (compteur < = 5) faire //condition de poursuite

{ //début du bloc

ecrire (compteur) ; //traitement

compteur ⟵ compteur+1 ; //incrémentation du compteur

}  //fin du bloc

Fin

La variable compteur est initialisée avant la boucle pour que la condition de poursuite (compteur < = 4) soit valide et cette dernière est examinée en premier lors de l'exécution de la boucle. Arrivée à la fin du bloc, l'exécution de l'algorithme reprend au niveau de *tant_que*, pour évaluer à nouveau la valeur booléenne de la condition.

Tableau explicatif de *tant_que* sur base de l'exemple

| Compteur | Condition :<br>(compteur < = 4) | Condition de contuiité |
|---|---|---|
| 1 | Initialisation : avant de rentrer dans la boucle | |
| 1 | 1< = 4 : vrai | Entrer dans la boucle |
| 2 | < = 4 : vrai | Encore un tour |
| 3 | 3< = 4 : vrai | Encore un tour |
| 4 | 4< = 4 : vrai | Encore un tour |
| 5 | 5< = 4 : faux | Sortir de la boucle |

Il ya toujours plusieurs manières d'écrire la condition de poursuite de la boucle pour obtenir exactement le même résultat.

N.B : il est plus naturel de se demander « quand la boucle s'arrête-telle ? » que de déterminer la condition de continuité « tant_que quoi la boucle continue-t-elle ? ».

Pour écrire une boucle, il faut prendre l'habitude de :

- Chercher la condition d'arrêt :
- Ecrire sa négation à l'aide du tableau de correspondances des conditions d'arrêt suivant :

| Logique d'arrêt | = | ≠ | > = | < | > | < = | ET | OU |
|---|---|---|---|---|---|---|---|---|
| Logique de continuité | ≠ | = | < = | > | < | > = | OU | ET |

- Etapes d'écriture d'une boucle

Pour écrire une boucle, il est obligatoire de suivre ces 3 étapes :

- L'initialisation des variables du compteur, et en général du bloc, avant d'entrer dans la boucle
- La condition de poursuite : il existe toujours différentes conditions de poursuite qui sont toutes justes (équivalentes)
- La modification d'au moins une valeur dans la boucle (celle que l'on a initialisée précédemment) pour que la répétition exprime une évolution de calculs

N.B : C'est une erreur grave de négliger l'un des points précédents : on risque de ne entrer dans la boucle (la condition de poursuite est fausse dès le début) ou de ne pas pouvoir en sortir (il s'agit alors d'une boucle infinie).

- *La syntaxe des autres boucles*

Nous avions dit que la boucle la plus naturelle est la boucle tant_que. Elle sera utilisée systématiquement dans tous les algorithmes.

Mais les langages informatiques disposent d nombreuses syntaxe pour alléger l'écriture de programmes. Voici deux nouveaux types de boucles :

- *La boucle pour_faire*

La boucle pour_faire est la plus utilisée en programmation pour réitérer une exécution un nombre de fois connu à l'avance. Cette écriture est pratique puisqu'elle désigne l'ensemble de la boucle en une seule ligne : l'incrémentation (de 1) de la variable est sous-entendue à la fin de la boucle.

Exemple : affichage de nombres de 1 à 5

```
Algorithme autre_boucle

Variables : variable : entier ;

Debut

    Pour (compteur ⟵ 1) jusqu'à 5) faire

    {

        ecrire(compteur) ;

    }

Fin
```

Comment passer d'une écriture pour_faire à une écriture tant_que_faire :

| | |
|---|---|
| Algorithme boucle_tant_que_faire<br>Variables : compteur : entier ;<br>Debut<br>compteur ⟵ 1 ;<br>tant_que (compteur < = 5) faire<br>{<br>ecrire (compteur) ;<br>compteur ⟵ compteur + 1 ;<br>}<br>Fin | Algorithme boucle_pour_faire<br>Variables : compteur : entier ;<br>Debut<br>pour compteur⟵1 jusqu'à 5 faire;<br>{<br>ecrire (compteur) ;<br>}<br>Fin |

- *La boucle faire tant_que*

La boucle faire_tant_que effectue l'évaluation de la condition booléenne après avoir effectué le premier tour de boucle. Dans certains cas, les algorithmes s'écrivent avec moins de lignes en utilisant ce type de boucle. Faites néanmoins attention à la condition de continuité.

```
Algorithme boucle _ faire _ tant_  que
Variables : compteur : entier ;
Debut
        compteur  ←—  1 ; //initialisation
      faire                        //condition de poursuite
         {
            ecrire (compteur) ;  //traitement
                compteur       ←—  compteur + 1 ; //incrémentation du compteur
           }
       tant_que (compteur < = 4) ; // attention à la condition
Fin
```

L'utilisation efficace de cette boucle est nécessaire et suffisante pour savoir programmer. Elle n'est pas facile à maitriser, et c'est donc la seule qui sera utilisée dans la plus part de programmes.

### D. **Les boucles imbriquées**

Toute structure peut apparaitre au sein d'une autre. La notion de boucles imbriquées, n'apporte rien de nouveaux par rapport à celle de boucles déjà vue dans les pages précédentes, mais il est nécessaire de découvrir ce qu'elles ont de différence.

Autrement dit, il n'y  a qu'un bloc d'instructions à répéter lors d'une boucle. Mais, le bloc peut être lui-même composé d'une ou plusieurs boucles. On parle alors de *boucles imbriquées.*

Revenons à cet exemple de la saisie de notes, mais ici extrayons la meilleure note. Ajoutons comme contrainte supplémentaire qu'une note est comprise entre 0 et 20. Si ce n'est pas le cas, l'algorithme doit prévenir l'utilisateur pour qu'il recommence la saisie.

Etudions déjà la saisie d'une note comprise entre 0 et 20. La boucle s'arrête quand la note saisie est comprise entre 0 et 20 (note > = 0) ET (note < = 20), la condition de continuité s'écrit donc ((note < 0) OU (note >20)).

```
Algorithme note

Variables : note : entier ;

Debut

    Ecrire (''Entrer une note :'') ;

    Lire (note) ; //utilisateur entre la note

Tant_que ((note < 0) OU (note >20)) faire

    {

      ecrire(''Vous avez fait une erreur, essayez encore :'') ; //message d'erreur affiché

      lire (note) ; //on recommence la saisie.

    }

Fin
```

Intégrons ce bloc dans la saisie de 5 notes pour déterminer la plus grande :

```
Algorithme note
Variables : compteur, max, note : entier ;
Debut
    Ecrire (''Entrer une note :'') ;
    Lire (note) ; //utilisateur entre la note
Tant_que ((note < 0) OU (note >20)) faire
    {
      ecrire(''Vous avez fait une erreur, essayez encore :'') ; //message d'erreur affiché
      lire (note) ; //on recommence la saisie.
    }
max ←——— note ;
compteur ←——— 1 ;
tant_que (compteur <5) faire //condition de poursuite
{                                    //corps de la boucle
    ecrire (''Entrer une note :'') ;
    lire(note) ; //taper la note
tant_que (compteur <0) OU (note >20) faire
{
```

```
        ecrire (‘Erreur, essayez encore’’) ;
        Lire(note) ;
Si (max <  note) alors
{
        max ←————note ;
}
   compteur  ←———— compteur+1 ;
}
ecrire (‘La note la plus grande est, max’’) ;
Fin
```

Cet exemple pourrait être judicieusement réécrit avec une boucle faire_tant_que vue précédemment.

*N.B : L’erreur la plus fréquente dans les boucles imbriquées consiste à oublier d’initialiser l’indice de la boucle intermédiaire avant chaque passage*.

## E. **Exercices pratique A faire avec les étudiants**

1. Faites lire à l’utilisateur 5 nombres entiers et afficher le plus grand d’eux.
2. Faire un programme qui attribue l’état civil à un individu entré au clavier par l’utilisateur.
3. Algorithme qui permet de saisir les éléments d’un tableau de dimension 8 grâce à une boucle.

4. Nous désirons écrire un programme qui travaille avec un damier de 10 cases sur 10 contenants des entiers. Chaque élément est alors repère par deux indices : le numéro de la ligne et d la colonne. Ecrire l'algorithme permettant de mettre à Zéro tous les éléments du damier.

5. Ecrivons un algorithme qui détermine la position d'une valeur dans un tableau d'entiers. Le tableau est initialisé par des valeurs lues au clavier et l'utilisateur cherche ensuite une valeur. Faire l'algorithme déterminant la position du premier élément du tableau (même s'il y en a plusieurs) ou il renvoie -1 si le tableau ne possède pas cette valeur.

6. Ecrire un algorithme qui demande à l'utilisateur de saisir une série de nombres entiers entre 0 et 20 et les stocke dans tableau de 50 éléments. La saisie s'arrête si l'utilisateur saisit -1 ou si le tableau est complet. Sinon, à chaque erreur de saisie, l'utilisateur doit recommencer.

7. Ecrire un algorithme qui permet d'afficher les tables de multiplications de 1 à 10.

# *Chap. III : Les fonctions*

Tout au début de ce cours, nous avions déjà eu à étudier les fonctions écrire et lire pour saisir et afficher des valeurs.

Une fonction fournit un service dans un algorithme isolé. Lorsque votre algorithme doit effectuer plusieurs fois la même tâche, il est judicieux d'isoler cette tâche dans une fonction e de l'appeler aux moments opportuns : votre algorithme n'en sera que plus facile à écrire et à modifier.

L'intérêt de l'utilisation de fonctions est double : il existe dans tous les langages informatiques des bibliothèques de fonctions associées à de domaines de traitements particuliers (traitement des fichiers, des images, des animations, etc. pour maitriser un langage, un programmeur doit connaitre et utilise les bibliothèques de fonctions.

La maitrise de fonctions est une étape nécessaire à la compréhension ultérieure de la notion d'objet (cfr. programmation orientée objet) et de méthode. Il est raisonnable d'être progressif dans l'étude d'un bloc de programme exécuté à l'extérieur d'un algorithme : la fonction est l'exemple le plus simple.

## *A.* **Les foncions simples**

- *Définition*

Par définition une fonction est un algorithme indépendant. L'appel (avec ou sans paramètres) de la fonction déclenche l'exécution de son bloc d'instructions. Une fonction se termine ***retournant ou non une valeur.***

Il faut aussi ajouter, qu'en programmation, il existe ce que nous appelons la ***procédure***, qui se définit comme étant une fonction qui ***retourne vide : aucune valeur n'est retournée.***

*Structure d'une fonction :*

Fonction nomDeLaFoonction (liste des paramètres) : typeRetourne

Debut

Bloc d'instructions ;

Fin

Dans l'exécution d'une fonction, 3 étapes sont nécessaire :

- Le programme appelant interrompt son exécution ;
- La fonction appelée effectue son bloc d'instructions. Dès qu'une instruction *retourne* est exécutée, la fonction s'arrête.
- Le programme appelant reprend alors son exécution

N.B : L'arrêt de la fonction a lieu lorsque son exécution atteint la fin du bloc d'instructions, ou lorsque l'instruction *retourne* est exécutée (avec ou sans valeur).

Le programmeur doit penser à concevoir et écrire des fonctions pour améliorer son programme. Il y gagnera sur plusieurs points :

- Le code des algorithmes est plus simple, plus clair et plus court. Dans un algorithme, appeler une fonction se fait en une seule ligne et la fonction peut être appelée à plusieurs reprises.
- Une seule modification dans la fonction sera automatiquement répercutée sur tous les algorithmes qui utilisent cette focntion.

- L'utilisation de fonctions génériques dans des algorithmes différents permet de réutiliser son travail et de gagner du temps.

- *Fonction sans valeur retournée*

Apprenons à écrire et utiliser une fonction simple qui doit afficher ''Bonjour''. Cette fonction ne retourne pas de valeur :ceci est signalé en précisant qu'elle retourne vide.

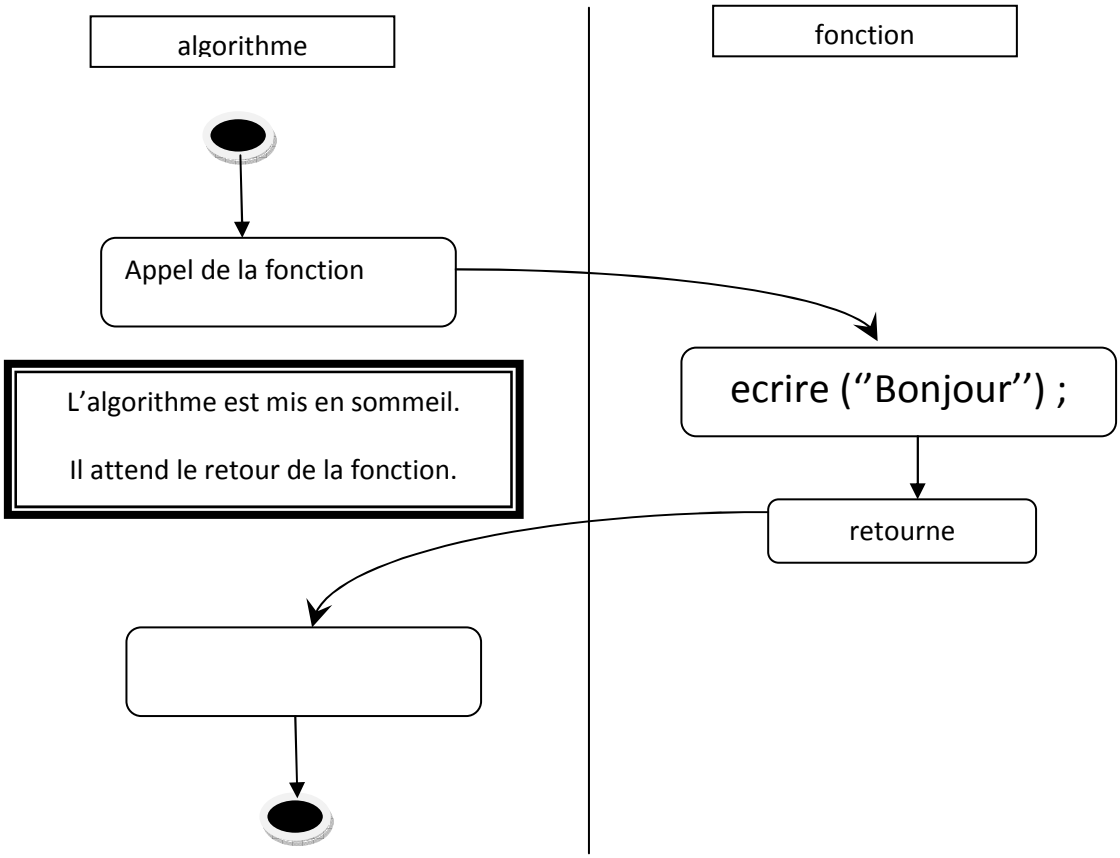
```
fonction affichermessage() :vide
Debut
    ecrire (''Bonjour'') ;
    retourne ;
Fin
```

Une fonction se termine toujours par l'instruction ***retourne.*** Cette fonction effectuera les instructions situées entre Debut et Fin.

Ecrivons un algorithme qui appelle la fonction affichemessage ().

```
Algorithme appelle_fonction
Debut
    affichermessage() ;
Fin
```

Schématiquement nous avons ceci : Passage de l'algorithme à la fonction.




Imaginons un autre algorithme qui appelle 10 fois la fonction affichermessage () :

Algorithme appelle_fonction_10_fois

Variable : indice : entier ;

Debut

Indice ← 0 ;

Tant_que (indice <10);

{

affichermessage () ;

Indice ⟵ indice +1 ;

}

Fin

N.B : Pour faciliter la lecture des algorithmes, il convient de respecter des règles (ici inspirées du langage Java) pour nommer les fonctions :

- Le nom d'une fonction commence par une minuscule ;
- Le nom d'une fonction ne comporte pas d'espace ;
- Si le nom de la fonction est composé de plusieurs mots, faire commencer chacun d'eux par une majuscule et ne pas faire figurer de traits d'unions.

- *Fonction avec valeur retournée*

De part la définition, une fonction retourne une valeur au programme appelant. Cette valeur est unique. Le retour de la valeur signifie l'arrêt de la boule.

Pour comprendre cette notion, partons d'un exemple. Introduisons une fonction qui permet de lire une note entre 0 et 20.

fonction lireNote () : entier

variables : note : entier ;

Debut

ecrire (‘’ Entrer une note :’’) ;

lire (note) ;

Tant_que(note < 0) ou (note > 20) faire

{

ecrire (‘’Vous avez commis une erreur, essayez de nouveau’’) ;

lire (note) ;

}

retourne (note) :

Fin

La fonction *lireNote()* retourne une valeur entière à la fin de son exécution. L’instruction *retourne* indique la fin immédiate de la fonction et le retour dans le programme appelant.

### *B.* l’environnement de données

- *Les paramètres*

Le programme appelant doit donner à certaine fonction des valeurs pour effectuer ses calculs. La fonction associe à ses valeurs des variables afin de les manipuler, ce sont les paramètres de la fonction.

Par définition un paramètre est une variable locale à une fonction. Il possède dès le début de la fonction la valeur passée par le programme appelant.

Partons d’un exemple pour comprendre cette notion de paramètres de fonction : fonction maxDe2Valeurs qui retourne le maximum de deux valeurs passés en paramètres. Cette fonction doit retourner une valeur entière : celle-ci est calculée en tenant compte des deux valeurs passées en paramètres.

```
fonction maxDe2Valeurs (p1 : entier,  p2 : entier) : entier
Variables : resultat : entier ;
Debut
    Si (p1 < p2) alors
    {
        resultat⟵p2;
    }
    sinon
    {
        resultat⟵p1;
    }
    retourne (resultat) ;
Fin
```

Cette fonction effectuera les opérations situées entre Debut et Fin. Et il faut éviter d'utiliser l'instruction *retourne* en milieu de la fonction ; la lisibilité est alors moins facile.

N.B : Dans le cas d'une fonction qui comporte plusieurs instructions retourne, faites attention : la fonction se termine immédiatement lors de la première instruction *retourne* exécutée.

- *les données d'une fonction*

Dans l'utilisation et l'écriture d'une fonction, la plus grande difficulté est de comprendre l'ensemble de données auxquelles la fonction a accès.

Un environnement de données, appelé encre espace d'adressage, correspond à l'ensemble de variables associées exclusivement à un algorithme ou une fonction.

Une variable définie dans un algorithme (respectivement dans une fonction), existe uniquement le temps limité de l'exécution de l'algorithme (respectivement de la fonction). Peu importe le nom des variables définies dans la fonction pour pourvoir l'utilisée.

Ainsi, un programmeur ne donne jamais le nom des variables internes à une fonction dont il est auteur. Il est tout à fait possible que 2 variables, l'une déclarée dans le programme appelant et l'autre déclarée dans la fonction, portent le même nom. Elles peuvent être du même type ou non, peu importe, puisqu'elles sont utilisées de manière différente dans des environnements de données différents.

- *Les paramètres et les variables*

La plupart du temps, l'exécution d'une fonction est paramétrable grâce à des valeurs qui lui sont passées. Les paramètres sont des variables, comme nous l'avons dit, de la fonction. Il est donc faux de vouloir les redéfinir dans la zone de déclaration des variables.

Une fonction peut accéder à deux types de données :

- Les **paramètres** : dont les valeurs sont connues dès le début de la fonction. Les valeurs sont passées en paramètres. Il est inutile

de nommer les paramètres avec le même nom que les variables utilisées lors de l'appel de la fonction.

- Les **variables** : appelées variables locales, elles sont définies dans le bloc de déclaration des variables.

Au cours de l'exécution d'une fonction, les variables définies dans le programme appelant sont inconnues : aussi bien leur nom que leur valeur.

La valeur retournée est unique. Il est impossible pour une fonction de retourner plusieurs valeurs, mais également de modifier directement une variable du programme appelant (sauf la fonction lire).

- *Techniques*

La signature d'une fonction décrit les éléments qui permettent de l'appeler correctement, entre autres :

- Le nom de la fonction
- Le type (et ordre) de paramètres
- Le type de la valeur retournée.

Un programme qui souhaite utiliser une fonction n'a pas besoin de connaitre le corps de la fonction (situé entre Debut et Fin), ni même le nom ou les types des variables internes à la fonction ; mais seulement les caractéristiques nécessaires à son utilisation, donc sa **signature.** Il s'agit en fait de la **carte d'identité** de la fonction.

Bien sur le programmeur doit connaitre l'action de la fonction qu'il utilise en plus de savoir l'appeler.

N.B : Lors de l'utilisation d'une fonction, il faut éviter :

- D'oublier les parenthèses
- De ne pas respecter le type de retour
- De ne pas respecter le type de paramètre
- De ne pas réécrire à chaque fois une fonction qui existe déjà
- De croire que la fonction peut modifier la variable du programme.

Lors de l'écriture d'une fonction, il faut aussi éviter de :

- Donner le même nom à un paramètre et à une variable
- Placer plusieurs ***retourne*** consécutifs dans la fonction
- Vouloir retourner plusieurs valeurs
- Oublier de retourner la valeur ou retourner une valeur du mauvais type
- Penser qu'en modifiant la valeur d'un paramètre, celui-ci sera modifié dans le programme appelant.

### *C. Exercices*

1. Ecrire une fonction qui retourne le plus grand de deux entiers passés en paramètres. Même exercice avec 3 entiers.
2. Ecrire une fonction qui retourne la somme de deux entiers somme (entier, entier) retourne entier et un algorithme qui l'utilise. Expliquer à travers cet exemple la notion de variables locales.

3. Ecrire un programme qui demande à l'utilisateur de deviner un nombre entre 1 et 1000. A chaque proposition, le programme indique si le nombre à trouver est inférieur ou supérieur à celui saisi.

# Chap. II : Pseudo-codes – Ordinogrammes

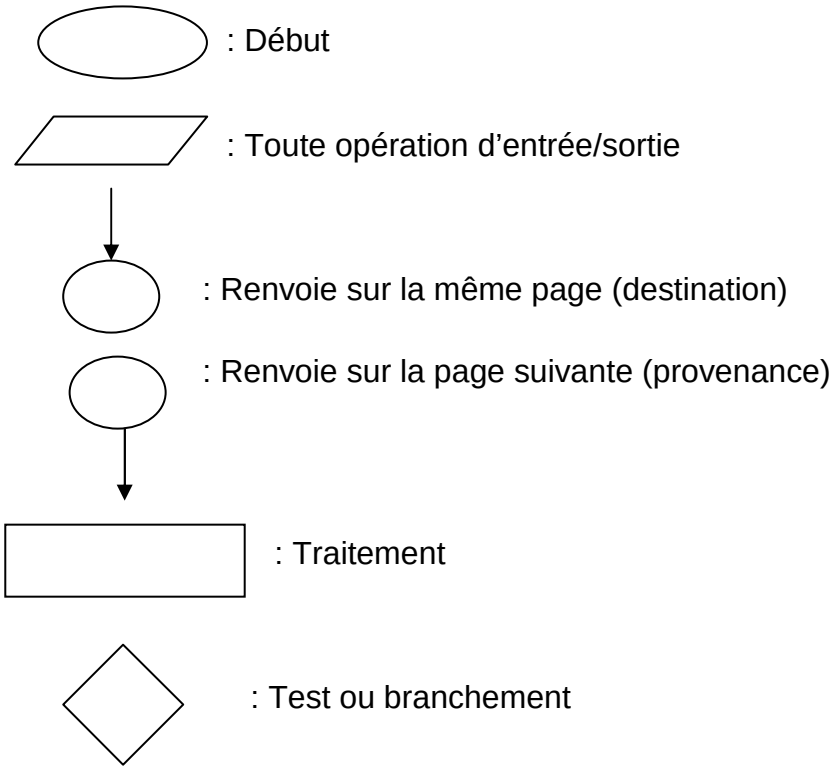
## A. Les ordinogrammes

Nous avons déjà eu à survoler les terminologies sur la représentation d'un algorithme dans les chapitres précédents.

Historiquement, plusieurs types de notations ont représenté des algorithmes. Il y a eu notamment une représentation graphique, avec des carrés, des losanges, etc. qu'on a qualifiée d'**organigrammes**. Aujourd'hui, cette représentation est quasiment abandonnée, pour deux raisons. D'abord, parce que dès que l'algorithme commence à grossir un peu, ce n'est plus pratique du tout. Ensuite parce que cette représentation favorise le glissement vers un certain type de programmation, dite non structurée que l'on tente au contraire d'éviter.

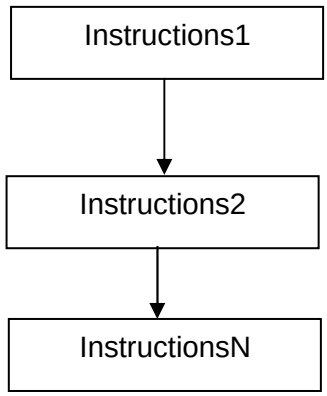
L'élaboration d'un algorithme est une notion que nous devons maîtriser dans les plus détailles et surtout que toute personne qui veut résoudre un problème complexe pensera à le ramener à des problèmes plus simples auxquels elle apportera une solution suivant un ordre logique.

Un ordinogramme est constitué de :



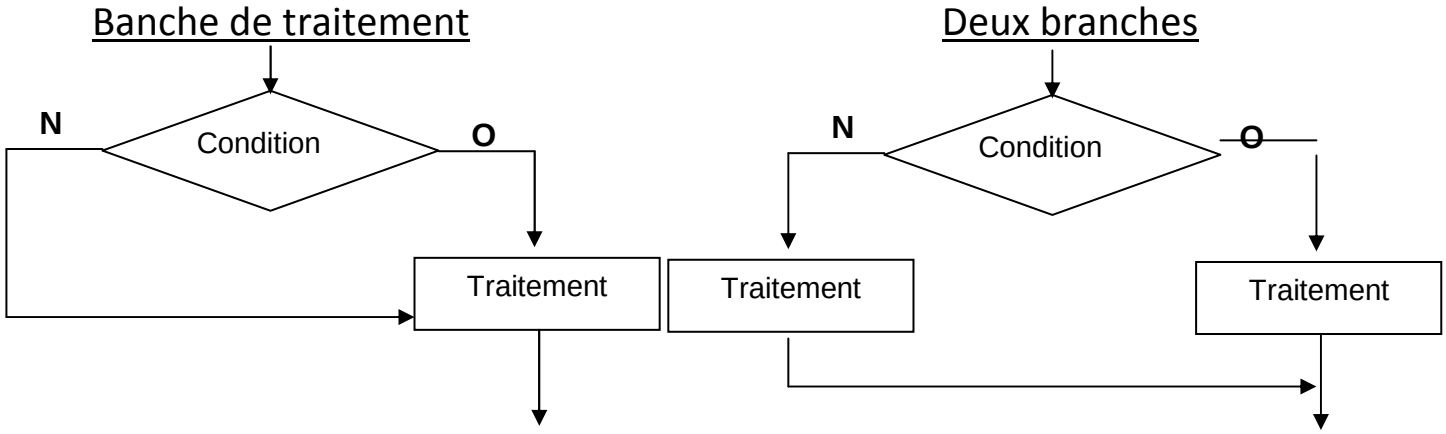

Toute structuration d'un programme doit se baser sur des traitements décomposables en faisant recourt à différentes structures comme :

- la structure linéaire ou séquentielle : qui est formée d'une suite d'instructions dans un ordre séquentiel sous la forme.




- La structure linéaire : qui permet d'effectuer un choix entre deux traitements, c'est-à-dire entre deux décisions pour résoudre un problème donné. Sur la branche de choix, il y a une

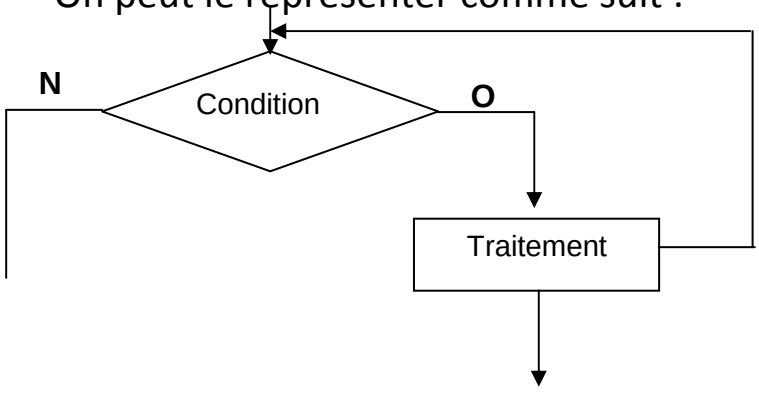
condition qui indique le chemin de choix qui dépendra de la réponse à la question (forme booléenne).




Si le traitement est juste, on passe au traitement envisagé, et dans le cas contraire, on fait un autre traitement correspondant à la réponse contraire ou simplement on va à la fin.

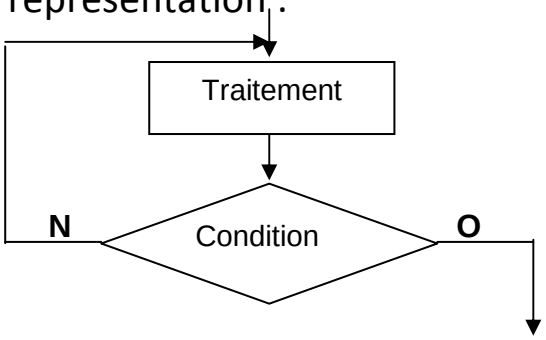
- La structure répétitive ou itérative : ici on fait allusion à la notion de boucle ; quand un traitement est appliqué à un objet plusieurs fois nous retiendrons ce qui est fréquemment utilisé dans la programmation. On peut trouver la boucle **DO While** <condition de continuation> ; c'est une itération qui effectue le teste avant le traitement.

On peut le représenter comme suit :




Tant que la condition sera respectée, on fera le même traitement avec la même formule sur les données différentes. On vérifie si on est à la fin pour terminer ou exécuter d'autres instructions avant de terminer

Une autre itération utilisée est celle de **DO Until**, qui consiste à répéter le traitement jusqu'à une condition d'arrêt. Elle a cette représentation :




Il faut retenir qu'il existe deux types d'algorithmes selon leur structure : les algorithmes récurrents et non récurrents (ceux qui appartiennent à la classe des algorithmes non récurrents linéaires et ceux qui qualifient les algorithmes non récurrents ramifié).

La notion de récurrence a beaucoup d'importance car elle permet de canaliser et d'économiser les directives dans un programme. Avec cette notion, on fait chaque fois recourt à un procédé itératif (répétitif) pour exprimer l'algorithme. L'algorithme récurrent introduit la nécessité d'intégration qui s'applique à plusieurs choses à la fois.

Pour réussir cette itération, l'algorithme récurrent doit introduire quatre notions utiles à connaître quelque soit le langage de programmation à utiliser (notion à expliquer dans la suite du cours) :

- la notion d'initialisation
- la notion de compteur
- la notion d'affectation
- la notion de boucle.

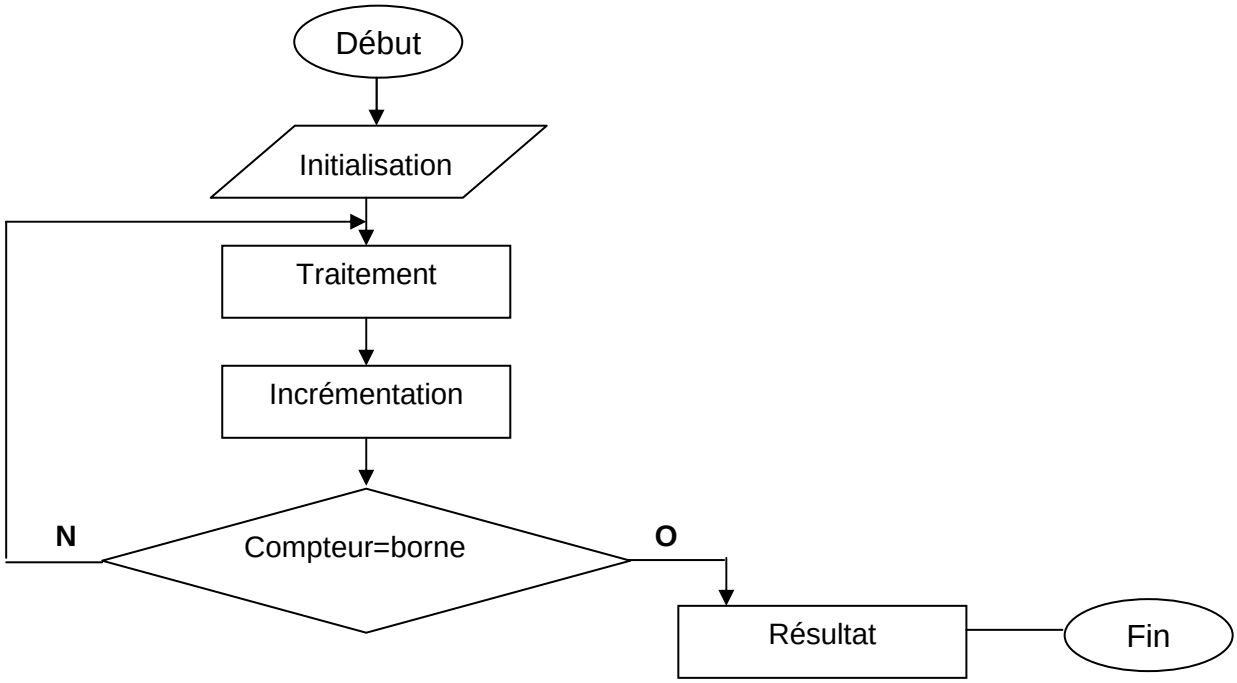
Pour comprendre bien ces notions servons nous de cet exemple : désignons **S** la zone dans laquelle sera arrangée les

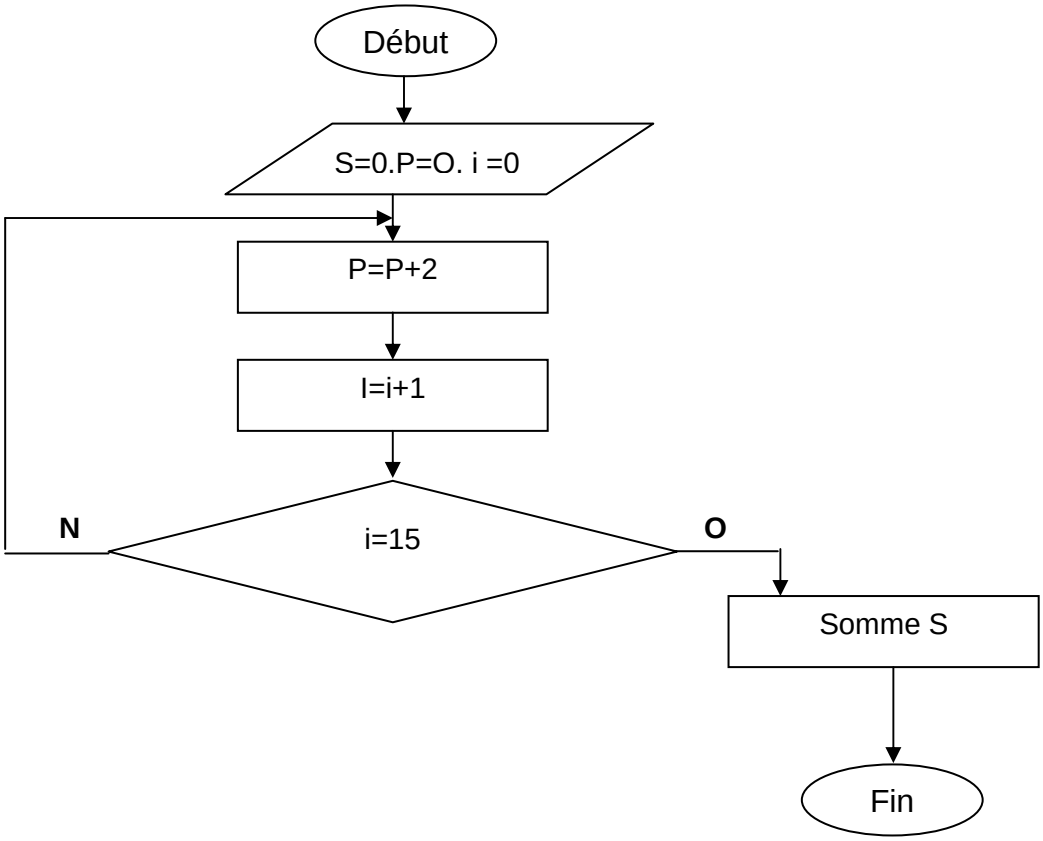
résultats d'une quelque opération ; et **i** la zone qui servira à compter les nombres pairs (compteur) à l'issue de chaque itération. Désignons par **P** le nombre pour calculer chaque itération. Quelles sont les natures de **S, i, P** ? Ce sont des variables.

La première chose, nous avons réservé des zones de mémoires, puis nous devons par l'initialisation de ces variables. Il faut une valeur initiale à chaque variable de calcul avant de commencer l'itération. Pour la somme de valeur initiale égale à Zéro (S=0) et les produits sont initialisés à un (P=1). A **i,** le compteur peut donner n'importe quelle valeur initiale mais il est préférable de l'attribuer soit 0, soit 1 comme valeur initiale. Dans notre exemple, prenons le pas de 2. D'où, P=P+2. Le signe d'égalité signifie que nous sommes d'affectés les valeurs dans la mémoire du nouveau résultat qui prend la place de l'ancien dans la zone réservée.

Nous incrémentons, dans la boucle i=i+1, d'une unité de compteur i afin de savoir par le test si on a atteint le nombre indiqué qui le pair. Le compteur permettra de ne pas dépasser la fin. Il faut contrôler cette incrémentation en comparant **i** à la borne (ici prenons la borne de 15), et si **i** dépasse 15, nous devons imprimer le résultat et mettre fin à notre algorithme.

Voici la résolution par l'ordinogramme de cet algorithme :






**B. Travail pratique :**

1. Dans une boutique de la place, si un client achète plus de 200 cartons de savons, on lui attribue une remise de 15% ; dans le cas contraire, on lui donne une remise 2%.
2. Une société appartenant à Mr MUSHAGALUSA dispose d'un fichier employé sur un support en raison d'un enregistrement par employé. Chaque enregistrement comprend le numéro matricule, le nom ; le post nom, le prénom, le sexe et le lieu de naissance de l'employé. On demande de résoudre cet algorithme en permettant de connaître les employés dont le sexe est masculin et dont la le lieu de naissance est Bukavu.
3. Voici les conditions de règlement au sein de l'entreprise TAMBWE : « bénéficie d'un escompte de règlement tout client qui règle par chèque ou par virement dans sept jours de la réception de la facture dont le montant est supérieur à 500. par contre pour

toute facture supérieure à 800, le délai ci-dessus est porté à quinze jours. Dans tous les cas, le client n'a droit à un escompte que si son compte ne présente de solde débiteur à l'émission de la facture.

**4.** Soit ces équations :

ax2+bx+c=0

ax+b=0.

Résoudre ces deux algorithmes.

**5.** Ecrire un programme qui permet de donner le temps de l'eau selon sa température en choisissant selon les trois réponses possibles : solide, liquide, gazeuse.

**6.** calculez et imprimez le carré de vingt premiers nombres impairs >32

**7.** Dans une librairie de Kigali, on veut promouvoir la vente d'une promotion. Tous les livres de cette collection sont vendus à un prix P. une remise de 12% est accordée à tout achat >450 ; on souhaite calculer et effectuer le prix net de la facture

**8.** Voici les extraits des instructions données au magasinier MILENGE de l'entreprise SOHB : « quand la commande d'un article donné ne dépasse pas la quantité maximum autorisée et quand le crédit du client est satisfaisant, prépare le bon d'expédition et avertie le service de livraison. Dans les autres cas, refuse la commande ; toutefois, si le crédit est satisfaisant, place la commande en attente. Si la quantité disponible ne permet pas de satisfaire le commande entièrement, place la commande en attente. Quand la quantité dépasse le maximum, avertie le service des achats. »

**9.** Un magasin de vente mixte trouve bon de fixer le prix de vent promotionnel. Le client qui achète X appareils est facturée au P1,

s'il est grossiste ; dans le cas où il achète au moins n unité, on lui accorde une réduction de 10%.

**10.** Vous disposez d'un fichier nommé Paie sur votre clé USB ; chaque enregistrement contient : le numéro matricule, le nom, le code sexe, le nom de l'atelier, le nombre d'heures prestées. Le code sexe sera M ou F. Calculer le salaire total brut de tous les employés de cette entreprise sachant que le salaire horaire est de 5$ pour les hommes qui ont travaillé au moins 20h et de 4$ pour les femmes qui ont travaillé plus de 15h et 3$ pour les autres.

**11.** L'école primaire NGEVESISE passe à l'inscription de nouveaux élèves. Les enregistrements suivants sont demandés à tous les élèves : nom, post nom, prénom, nom du père et de la mère, profession des parents et école de provenance. Tous les élèves de sexe féminin paient 8$ comme frais d'inscription (FI) s'ils sont du cycle d'orientation et 10 $ pour ceux du cycle long. Les élèves de sexe masculin paient 12$ quelque soit leur cycle. La condition sine qua non pour être admis dans cette école est le dépôt d'un dossier complet et conforme aux conditions d'inscription. Calculer le montant global des FI.

**12.** La boutique SOURIRE DU MATIN voudrait promouvoir la vente de ses articles. Deux catégories d'article sont mis à la disposition de tout client quelque soit sa catégorie puisque cette boutique est spécialisée dans la vente mixte : les articles A1 pour tout client grossiste pouvant bénéficier d'un escompte de 4% sur le PA des commandes <= 400 ; les articles A2 pour tout client grossiste ne bénéficiant pas de l'escompte et dont la quantité est >200. il faut retenir que tout détaillant jouit de l'escompte de 4% sur l'article A2 s'il a passé une commande >500.

**13.** résoudre l'algorithme qui attribue l'état civil à un individu entré au clavier par un utilisateur.

**14.** Calculer et imprimer la somme de carré de 15 premiers nombres pairs compris entre 0 et 30.

**T.D : Résoudre les algorithmes ci dessous.**

## C. Les Pseudo-codes

Dans les notes précédentes, nous avions eu à définir le concept pseudo code. Comme le nom l'indique, un pseudo code est un faux (= pseudo) code.

Tout programme à une logique à suivre et ceci permet toujours de résoudre l'algorithme cachant un problème donné dans ce programme.

Sans le savoir, nous avons déjà abordé cette notion.

Tout au long de chapitres précédents, tous les algorithmes faits sont de pseudo codes, vu qu'ils sont conçus dans un langage compréhensible par l'homme uniquement.

Exemple :

```
Algorithme note

Variables : note : entier ;

Debut

    Ecrire (''Entrer une note :'') ;

    Lire (note) ; //utilisateur entre la note

Tant_que ((note < 0) OU (note >20)) faire

    {
```

ecrire(‘’Vous avez fait une erreur, essayez encore :’’) ; //message d’erreur affiché

lire (note) ; //on recommence la saisie.

}

Fin

**Alors point n’est besoin d’y revenir.**

# *Chap. IV : Du langage algorithmique vers le* langage Visual basique for application (VBA)

## A. Présentation

Visual Basic s'est développé à partir du langage BASIC (Beginner's All-purpose Symbolic Instruction Code, Larousse : " Langage de programmation conçu pour l'utilisation interactive de terminaux ou de micro-ordinateurs ", 1960).

Le but du langage BASIC était d'aider les gens à apprendre la programmation. Son utilisation très répandue conduisit à de nombreuses améliorations.

Avec le développement (1980-1990) de l'interface utilisateur graphique (Graphical User Interface - GUI) de Windows Microsoft, BASIC a naturellement évolué pour donner lieu ‡ Visual Basic (1991). Depuis, plusieurs versions ont été proposées, Visual Basic 6 est apparue en 1998.

Visual Basic pour Applications est le langage de programmation des applications de Microsoft Office. VBA permet d'automatiser les tâches, de créer des applications complètes, de sécuriser vos saisies et vos documents, de créer de nouveaux menus et de nouvelles fonctions pour booster efficacement votre logiciel.

Avant qu'Excel n'utilise ce langage de programmation, le logiciel utilisait son propre langage de programmation et une application était appelée « macro ». Ce terme est resté, mais une macro Excel réalisée en VBA n'est rien d'autre qu'une procédure telle qu'elles sont réalisées sous VB.

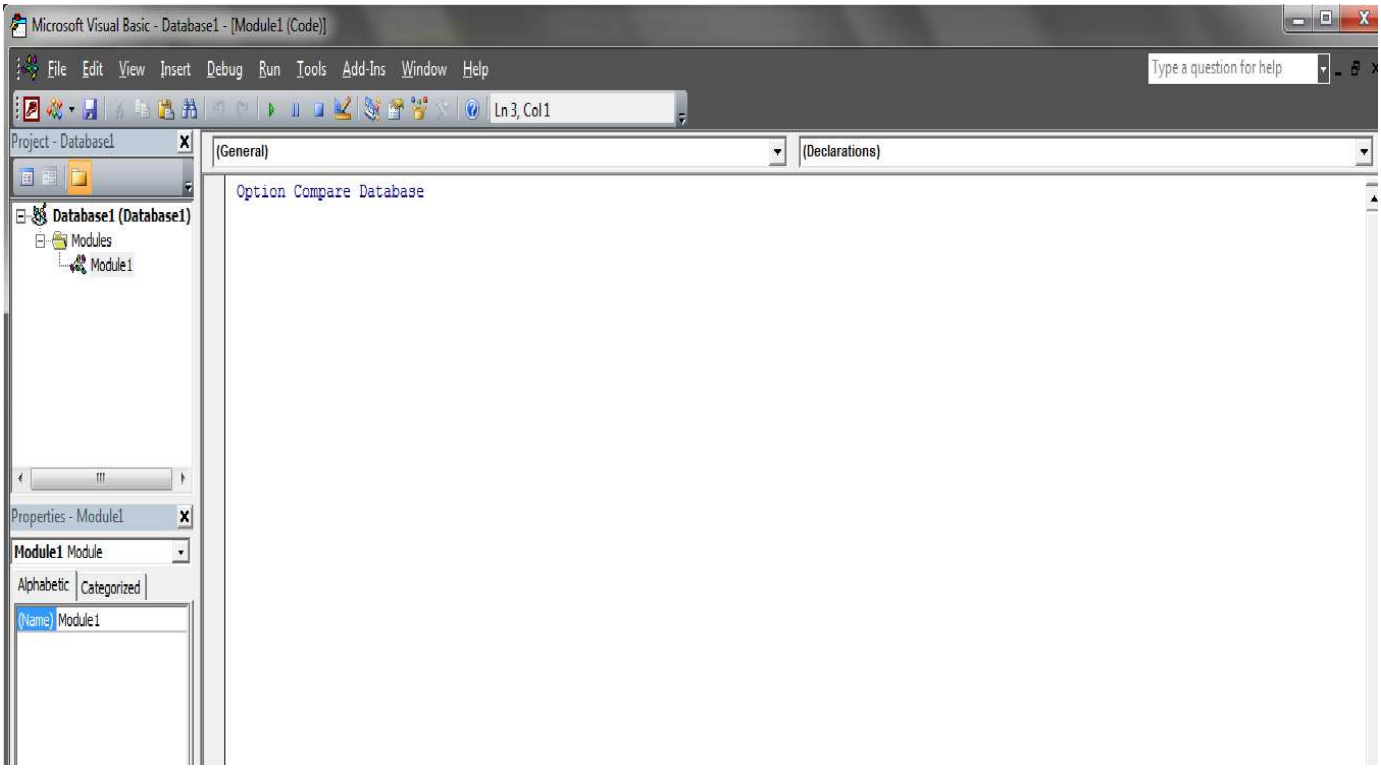
Un programmeur sous VBA n'a aucun problème pour passer à VB et vice-versa. Le langage VBA est accessible à tous. Cependant, une bonne connaissance d'Excel est nécessaire avant de se lancer dans la création d'application.

En effet, il est important de bien maîtriser les principaux objets que manipule VBA, comme les objets Workbook (classeur), Worskheet (Feuille de calcul), Range(plage de cellule), etc.…

## B. Editeur sous Office 2007

La première chose à faire est de lancer Access 2007 puis de créer une base de données dans un emplacement donné.


La seconde de chose est de situer dans la partie de l'objet Access appelé *Module*.




Access basic est un langage de programmation qui a été développé spécialement pour Access sur la base de Visual Basic de Microsoft.

Comme son nom l'indique, Access Basic ressemble plus au Basic qu'à d'autres langages de programmation.

Voici ce qu'Access Basic vous permet de faire :

- Ecriture de fonctions personnalisées qui pourront être utilisées à loisir dans les expressions Access
- Automatisation du traitement d'objets et de données dans une base de données
- Maintenance aisée des applications créées par l'utilisateur
- Modification d'enregistrements isolés
- Création d'applications de bases de données complexes concernant des cas spéciaux

VBA utilise le même langage que Microsoft Visual Basic. La différence entre VB et VBA est que VB est un ensemble complet qui permet de développer des applications indépendantes et librement distribuables alors qu'une application réalisée en VBA est complètement liée au logiciel sous lequel elle a été créée (une application VBA créée sous Access ne pourra pas se lancer sur un poste si Access n'est pas installé).

## C. Les variables

En développement, une variable est une donné définie dans un contexte donné ayant un type défini.

Autrement dit, une variable est la représentation d'une valeur au sens large du terme. Elles sont nécessaires pour stocker (conserver) une valeur dynamique et réutilisable.

- *Déclaration*

On appelle déclaration, le fait de définir ma variable avant de l'utiliser, dimensionnement : le fait de lui donner un type.

En VB, la déclaration de variables n'est pas obligatoire tant que ***l'option Explicit*** n'est pas activée. On peut aussi écrire la directive ***Option Explicit*** au début de la section des déclarations d'un module.

Syntaxe : **nom_variable As Type_variable**

N.B : On peut utiliser une seule instruction pour déclarer plusieurs variables en les séparant par une virgule, comme nous l'avons vu dans les chapitres précédents.

- *Les types de variables*

On recommande fortement de **déclarer** les variables utilisées dans un programme. Le type d'une variable est la détermination du genre de valeur que la variable peut contenir.

En VBA, toutes les variables possèdent le type ***variant*** par défaut, appelé parfois type ***universel***. Une variable de type variant peut contenir n'importe quelle valeur à l'exécution, à l'exception des chaines de longueur fixe.

Voici les types de variables en VBA :

- Integer : de **-32 768** à **32 767**
- Long : de -2 147 483 648 à 2 147 483 647
- Single : décimaux en simple précision : 39 chiffres significatifs
- Double: Décimaux en double précision: plus de 300 chiffres significatifs!
- String : de 0 à 65 535 octets
- Variant : de type nombre, texte ou objet selon l'affectation faite.
- Boolean : de type logique

Pour la lisibilité du code on peut les commenter après une ***apostrophe (') ou par le mot rem***

Exemples :

```
Dim Taux As Single ' Taux de la TVA

Dim Réponse As String ' Mot proposé par le joueur
```

Pour éviter tout problème il est préférable d'initialiser les variables déclarées.

Compteur = 0

Taux = 20,6

Le langage Basic utilise 7 types de données dont les plus utilisés sont le type String (chaîne de caractères), le type Integer (entier relatif) et le type Single (décimal).

- *Portée d'une variable*

Parfois appelée visibilité, elle définie les limites d'accessibilité d'une variable. Il existe plusieurs instructions de déclarations selon la portée désirée et la déclaration ne se fait pas au même endroit.

- Si une variable est déclarée au début de la procédure qui la manipule (Dim ou **Private**) elle n'est alors valide que pour cette procédure. L'existence et la valeur de la variable disparaissent avec l'instruction *End Sub*. Toute référence à cette variable en dehors de cette procédure provoquera une erreur de compilation.
- Si une variable est déclarée dans la section des déclarations d'un module elle est valide dans toutes les procédures du module.
- Une variable peut aussi être déclarée **Public** ou **Global** et sera alors valide pour toute l'application.

Exemple : Global MotCommun As String

La portée introduit la notion de durée de vie des variables. Les variables de niveaux modules sont dites permanente dans le sens

où elles existent tant que le code s'exécute. On les appelle aussi variables globales.

Les variables de niveau fonction n'existe que lorsque la fonction s'exécute et sont détruites quand la fonction se termine. Elles ne sont donc jamais accessibles en dehors de la fonction oû elles sont déclarées. On les appelle variables locales. Normalement, elles sont détruites en fin de fonction et perdent leur valeur, mais les variables statistiques (déclarées leur valeur, mais l'instruction Static) la conserve.

- *Constantes*

Ce sont des variables qui ne varient pas. Elles se déclarent à l'aide du mot *Const,* peuvent utiliser une instruction de porté et un type.

- *Opérateurs*

Ils servent à réaliser des opérations. Il existe relativement peu d'opérateurs en VBA, qui sont divisés en 5 familles :

- Les opérateurs mathématiques : +, -, *, /, \ (division entière, renvoie la partie entière), ^ (exposant), mod (modulo).
- Les opérateurs de comparaison : <, >, <=, >=, <>
- L'opérateur de concaténation: & permet de reunir 2 chaines.
- Les opérateurs logiques : And, Or, Xor
- Les opérateurs d'affectation : =

*N.B : contrairement à d'autres langages, VBA utilise le même opérateur = pour l'affectation et pour l'égalité.*

## D. Les procédures et fonctions

Une procédure (comme méthode dans le cas des objets), est un ensemble d'instructions s'exécutant comme une entité.

Tout code exécutable est obligatoirement dans une procédure. En VBA, une procédure se définit quand on la déclare. La définition est toujours constituée telle que :

- Un mot clé définissant la portée (si cette portée est omise, la procédure sera publique si elle est dans un module standard, privé si non)
- L'instruction *Sub*
- Le nom de la procédure
- La liste des arguments.

N.B : Un argument et aussi appelé *paramètre* et est une variable que l'on communique à la procédure. Les arguments se déclarent lors de la définition de la procédure.

Pour appeler une fonction ou une procédure depuis un point de code, il faut d'abord et avant tout que sa portée le permette, c'est-à-dire qu'elle soit visible depuis le point d'appel.

Ici, nous appellerons les fonctions ou procédures avec le mot Call suivi de la procédure ou de la fonction. L'appel ici est sous entendu que l'utilisation de la procédure ou de la fonction.

Ex : Private sub afficher()

Lorsqu'une procédure renvoie une valeur, on l'appelle ***fonction.*** Une fonction se définit comme une procédure sauf qu'on

utilise FUNCTION au lieu de SUB, et on précise le type renvoyé. Si le type n'est pas précisé, la fonction est de type variant.

Ex : Private function afficher() as decimal

Puis viendra la liste d'instructions et on terminera par *End Sub* ou *End Function* dans le cas de la fonction.

## E. Les tableaux

- *Structure d'un tableau*

On a souvent besoin de travailler sur un *ensemble de données.* On appelle tableau, un ensemble d'éléments indexés séquentiellement, ayant le même type de données. Un tableau peut posséder jusqu'à 60 dimensions en VB.

*Exemple* : les températures moyennes des 12 mois de l'année.

On pourrait déclarer **12 variables identiques** :

**Dim** Temp1, Temp2, Temp3, Temp4, … … …, Temp12 as Single

On dispose d'une structure de données appelée **Tableau** qui permet de conserver dans une seule "**entité**" plusieurs valeurs de même type.

Le *nom* du tableau est une variable qu'il est recommandé de préfixer par Tab. Le nombre de valeurs de types identiques est à déclarer entre parenthèses.

- *Déclaration d'un tableau*

**Dim nomTableau (dimension) As typeTableau**

*Déclaration d'un tableau à une dimension :*

**Dim** TabTemp(12) **As Single**

| Numéro | 1 | 2 | 3 | 4 | 5 | … |
|---|---|---|---|---|---|---|
| Température | 6 | 5,5 | 7 | 11,5 | 15 | … |

L'accés à la case numéro 3 se fait par TabTemp(3) qui vaut **7**.

*Déclaration d'un tableau à deux dimensions :*

**Dim** TabMajMin(1 to 2,65 to 90) **As String**

| Numéro | 65 | 66 | 67 | … | 89 | 90 |
|---|---|---|---|---|---|---|
| 1 | A | B | C | … | Y | Z |
| 2 | a | b | c | … | y | z |

L'accès à la case qui contient le petit "c" s'obtient grâce à la syntaxe suivante : TabMajMin(2,67) qui vaut "c".

- *Boucles en nombre défini*

Cette boucle est utilisée lorqu'on connaît à l'avance le nombre de fois qu'elle sera parcourue.

*Syntaxe* :

**For** *Compteur* **=** *Début* **To** *Fin* [**Step** *Incrément*]

*Instructions*

[ **... Exit For**]

[*Instructions*]

**Next** [*Compteur*]

Le test est effectué **au début** de la boucle.

La variable numérique Compteur est **incrémentée** à chaque fin de boucle du nombre indiqué par *l'incrément*. Si l'incrément n'est pas spécifié il est fixé à **1**.

Si la valeur de **Fin** est inférieure à la valeur de **Début** l'incrément est *négatif*. La valeur de **Compteur** peut être utilisée (par exemple pour numéroter le passage dans la boucle) *mais ne doit pas être modifiée* dans le corps de la boucle.

*Exemple* :

For i = 1 To 50

TabInitial(i) = 0 ‘ Initialisation de chaque case à 0

Next i

- *Remplissage d'un tableau*

Pour remplir un tableau on le balaye avec une boucle (notion à voir un peu bas) **For … To … Next** (car le nombre de cases est connu à l'avance).

*Exemple 1* :

```
Dim TabTemp(12) As Single

Dim Compteur As Integer

For Compteur = 1 To 12

    TabTemp(Compteur)=InputBox("Température N° " & Compteur)

Next Compteur
```

*Exemple 2* :

```
Dim TabTirageLoto(6) As Integer

Dim Compteur As Integer

For Compteur = 1 To 6

    TabTirageLoto (Compteur)=Rnd * 48 + 1

Next Compteur
```

## F. Les structures conditionnelles et boucles

On appelle structure décisionnelle la construction de code permettant de tester une ou plusieurs expressions afin de déterminer le code à excuter.

Si le programme doit exécuter un bloc d'instructions en *nombre prédéfini* on utilise la boucle **For … To … Next.**

Exemple :

```
For i = 1 To 49

        TabLoto(i) = i ‘chaque case contient son numéro

Next i
```

Si le nombre de passages dans la boucle est inconnu au départ, mais dépend d'une **condition** dont la réalisation est imprévisible cette structure n'est pas adaptée.

*Exemple 1 :*

Demander le mot de passe **tant que** la réponse n'est pas le bon mot de passe.

Demander le mot de passe **jusqu'à ce que** la réponse soit le bon mot de passe.

*Exemple 2 :*

Demander la saisie d'une note **tant que** la réponse n'est pas un nombre entre 0 et 20.

Demander la saisie d'une note **jusqu'à ce que** la réponse soit un nombre entre 0 et 20.

**Boucle tant que**

*Syntaxe première version* :

Do While *Condition*

*Instructions*

[... Exit Do]

[*Instructions*]

Loop

La condition est ici testée **au début** c'est à dire à l'entrée de la boucle.

Avec While (tant que) la boucle est répétée **tant que** la condition est **vraie**.

Si la condition n'est pas vraie au départ les instructions de la boucle ne sont pas exécutées.

*Exemple* :

Do While MotProposé <> MotDePasse

MotProposé = InputBox("Donnez votre mot de passe")

Loop

Cela présuppose MotProposé initialisé par une valeur autre que MotDePasse

(par exemple la valeur par défaut "").

*Syntaxe deuxième version* :

Do

*Instructions*

[... Exit Do]

[*Instructions*]

Loop While *Condition*

La condition est alors testée **à la fin** de la boucle.

Avec While (tant que) la boucle est répétée **tant que** la condition est **vraie**.

Les instructions de la boucle sont donc exécutées au moins une fois.

*Exemple* :

Do

MotProposé = InputBox("Donnez votre mot de passe")

Loop While MotProposé <> MotDePasse

Cet exemple ne présuppose aucune initialisation de MotProposé.

**Les boucles Tant que , jusqu'à**

*Syntaxe première version* :

Do while *Condition*

*Instructions*

[... Exit Do]

[*Instructions*]

Loop

La condition est ici testée **au début** c'est à dire à l'entrée de la boucle.

Avec while (tant que) la boucle est répétée **tant que** la condition n'est pas **vérifiée**.

Si la condition est vraie dès le départ les instructions de la boucle ne sont pas exécutées.

*Exemple* :

Do while MotProposé <> MotDePasse

MotProposé = InputBox("Donnez votre mot de passe")

Loop

Cela présuppose MotProposé initialisé par une valeur autre que MotDePasse

(par exemple la valeur par défaut : "").

*Syntaxe deuxième version* :

Do

*Instructions*

[... Exit Do]

[*Instructions*]

Loop Until *Condition*

La condition est alors testée **à la fin** de la boucle.

Les instructions de la boucle sont donc exécutées au moins une fois.

Avec Until (jusqu'à) la boucle est répétée **jusqu'à ce que** la condition soit **vraie**.

*Exemple* :

Do

MotProposé = InputBox("Donnez votre mot de passe")

Loop Until MotProposé = MotDePasse

Cet exemple ne présuppose aucune initialisation de MotProposé.

**Boucle For … Each … Next**

C'est une extension de la boucle For ... To … Next. Elle est utilisée pour parcourir les collections(ensembles).

*Syntaxe* :

For Each *Elément* In *Ensemble*

*Instructions*

[ ... Exit For]

[*Instructions*]

Next [*Elément*]

*Ensemble* est le plus souvent un tableau.

*Exemple* :

```
Dim TabHasard(100) As Integer

Dim Cellule As Integer

Dim Réponse As String

Randomize

For Each Cellule In TabHasard

                Cellule = Rnd * 100 + 1

Next

For Each Cellule In TabHasard

Réponse = Réponse & Cellule & "   "

Next

MsgBox (Réponse)
```

De façon plus explicite parmi les strcutures, nous avons : ***If….Then…Else*** (cette structure admet la syntaxe linéaire et la syntaxe de blocs), ***Elseif….Then*** (permet de gérer des tests consécutifs dans un bloc If).

Et parmi les boucles (ou traitements itératifs), nous pouvons avoir :

- Les traitements à nombres d'itérations finies
- Les boucles à test

➢ For…Next

Sa syntaxe est la suivante

*For compteur=Debut To fin Step pas*

*Traitement*

*ExitFor*

*Traitement*

*Next compteur*

Exemple : Factoriel d'un nombre

Private Function (Arg As Integer) As long

Dim compt As long

fact =1

*For compt =1 To Arg Step 1*

*fact= fact*compt*

*Next For*

End Funcction

Vous ne pouvez pas omettre le pas (Step) sans quoi la boucle ne sera pas exécutée. La clause de sorte positionne le curseur d'exécution sur la ligne qui suit l'instruction Next de la boucle For.

- Do…Loop

Les boucles Do…Loop, parfois appelées bboucles conditionnelles utilisent un test pour vérifier s'il convient de continuer les itérations :

VBA utilse 2 tests différents :

- While (tant_que) : qui poursuit les itérations tant que l'expression du test est VRAIE.
- Until (jusqu'à) : qui poursuit les itérations jusqu'à que l'expression du test soit exécutée.

La position du test a influencé sur le nombre d'itérations. Si le test est positionné en début de boucle, c'est-à-dire sur la même ligne que le Do, il n'y aura pas forcément exécution du traitement contenu dans la boucle, par contre s'il est positionné à la fin, c'est-à-dire sur la ligne que Loop, le traitement sera toujours exécuté au moins une fois.

Quelle que soit la constrction choisie, il est toujours possible d'utiliser la clause de sortie EXIT DO.

Voici les 4 formes de boucles conditionnelles.

1. Do while expression

Traitement

Exit Do

Traitement

Loop

2. Do

Traitement

Exit Do

Traitement

Loop While expression

3. Do until expression

Traitement

Exit Do

Traitement

Loop

4. Do

Traitement

Exit Do

Traitement

Loop Until expression

**G. Exercices : Confère les Annexes ;**

# Conclusion

Nous sommes à la fin de ce cours d'Algorithmique, qui d'une part nous a donné les notions de base relatives à la structure de programmation, et d'autre part, nous a aussi fait découvrir le Visual Basic comme premier langage dans le monde de programmation.

C'est ainsi, que dans ce cours, il a été question de survoler les divers chapitres ci après : ***Les variables, les structures de contrôle, les fonctions, les Pseudo-codes – Ordinogramme et afin du langage algorithmique vers le*** **langage Visual basique.**

A la fin de chacun de ces chapitres, une série d'exercices non corrigés faisait partie de différentes parties du dit chapitre et dont les correctifs se trouvent dans la partie Annexe pour permettre à l'étudiant de se familiariser tout d'abord avec un travail d'essai fait au début, avant d'obtenir la suite de la bonne solution.

Pour mettre en place ce support de cours, nous avons dû recourir à beaucoup d'ouvrages et cours, et cela pour ne permettre aux étudiants d'avoir de notions purement générales sur le premier pas en programmation.

Toute personne qui consultera ce support de cours, qu'elle soit avancée ou débutant dans la programmation, découvrira de notions si élémentaires, mais très nécessaires pour exercer le métier de programmeur.

Nous sommes ouverts à toute remarque ou suggestion relative à ce support ou à n'importe quelle partie de l'informatique et sachez, cher lecteur, que le travail humain n'atteindra jamais la perfection.

***Ir Olga K. KINYAMUSITU***

# Bibliographie

**Ouvrage :**

1. Claude Delannoy, *Initiation à la programmation*, E. Eyrolles, paris, Mars 1999
2. G.CLAVEL*, Introduction à la programmation*, MASSON, paris, 1993
3. R.C BACKHOUSE, *Construction et vérification de Programmes*, Masson, 1986
4. Sylvie TORMENTO, *Algorithmes-cours et exercices*, 1ere édition, Rome, Septembre 1992
5. Jacques C. et Irène Kowarski, *Initiation à L'algorithmique et aux structures de données*, 1ere édition, volume 1, DUNOD, Paris 19942
6. ULRICH MATTHEY et UTE MEISER, *Le livre de l'utilisateur Microsoft Access 2*, Editions Mricro Application, 1994

**Cours :**

1. Alexandre Meslé, *Algorithmique pour le BTS*, 12 mai 2009
2. *Algorithmique-programmation*, Université de Nice
3. Christophe DABANCOURT, *Apprendre à programmer*, 2ème édition Editions Eyrolles Paris 2008
4. CT Eurasme M., *cours d'Algorithmique*, ISIG 2006-2007, inédit
5. Cours EDC, Claire PELTIER, 2007
6. Formation VBA, Mchel TANGUY, 2006

# ANNEXES:CORRECTION EXERCICES