SUPPORT DU
COURS D'INFORMATIQUE
1 TCSM

DR. ABDELKLRIM MEZIANE
ENSSMAL
2009/2010

TABLE DES MATIERES

CHAPITRE 1 : LE CODAGE 3
1.1. LES ORDINATEURS SONT « BINAIRES »
1.2. LA NUMEROTATION DE POSITION EN BASE DECIMALE
1.3. LA NUMEROTATION DE POSITION EN BASE BINAIRE
1.4. LE CODAGE HEXADECIMAL
1.5. Conversion des entiers positifs
1.6. Arithmétique binaire
1.7. Mémoire de l'ordinateur et codage
1.7.1 Nombres entiers positifs
1.7.2. Codage des entiers relatifs en complément à deux
1.7.3. Codage des caractères
CHAPITRE 2 INTRODUCTION A L'ALGORITHMIQUE.................................... 9
2.1. Algorithme
2.2. ALGORITHMIQUE ET PROGRAMMATION
2.3. AVEC QUELLES CONVENTIONS ECRIT-ON UN ALGORITHME ?
CHAPITRE 3 LES VARIABLES... 11
3.1. A QUOI SERVENT LES VARIABLES ?
3.2. DECLARATION DES VARIABLES
3.2.1 Types numériques classiques
3.2.2 Type caractère
3.2.3 Type booléen
3.3. L'INSTRUCTION D'AFFECTATION
3.3.1 Syntaxe et signification
3.3.2 Ordre des instructions
3.4. EXPRESSIONS ET OPERATEURS
3.4.1 Opérateurs numériques
3.4.2 Opérateur alphanumérique
3.4.3 Opérateurs logiques (ou booléens)
CHAPITRE 4 LECTURE ET ECRITURE.. 14
4.1. INTRODUCTION
4.2. LES INSTRUCTIONS DE LECTURE ET D'ECRITURE
CHAPITRE 5 LES TESTS.. 15
5.1. DEFINITION
5.2. STRUCTURE D'UN TEST
5.3. QU'EST CE QU'UNE CONDITION ?
5.4. CONDITIONS COMPOSEES
5.5. TESTS IMBRIQUES
CHAPITRE 6 LES BOUCLES.. 18
6.1. LA BOUCLE TANTQUE
6.2. LA BOUCLE POUR
6.3. LES BOUCLES IMBRIQUEES

CHAPITRE 7 LES TABLEAUX..20
7.1. UTILITE DES TABLEAUX
7.2. NOTATION ET UTILISATION ALGORITHMIQUE
7.3. TABLEAUX DYNAMIQUES
7.4. TABLEAUX A DEUX DIMENSIONS
7.5. TABLEAUX A N DIMENSIONS
CHAPITRE 8 QUELQUES TECHNIQUES CELEBRES.....................................23
8.1. TRI D'UN TABLEAU : LE TRI PAR SELECTION
8.2. UN EXEMPLE DE FLAG : LA RECHERCHE DANS UN TABLEAU
8.3. TRI DE TABLEAU + FLAG = TRI A BULLES
8.4. LA RECHERCHE DICHOTOMIQUE
CHAPITRE 9 LES FONCTIONS PREDEFINIES...28
9.1. STRUCTURE GENERALE DES FONCTIONS
9.2. LES FONCTIONS DE TEXTE
9.3. TROIS FONCTIONS NUMERIQUES CLASSIQUES
9.4. LES FONCTIONS DE CONVERSION
CHAPITRE 10 LES FICHIERS...31
10.1. ORGANISATION DES FICHIERS
10.2. STRUCTURE DES ENREGISTREMENTS
10.3. TYPES D'ACCES
10.4. INSTRUCTIONS (FICHIERS TEXTE EN ACCES SEQUENTIEL)
10.5. STRATEGIES DE TRAITEMENT
10.6. DONNEES STRUCTUREES
10.6.1 Données structurées simples
10.6.2 Tableaux de données structurées
10.7. RECAPITULATIF GENERAL
CHAPITRE 11 PROCEDURES ET FONCTIONS...38
11.1. FONCTIONS PERSONNALISEES
11.1.1 Définitions
11.1.2 Passage d'arguments
11.2. PROCEDURES
11.2.1 Généralités
11.2.2 Le problème des arguments
11.2.3 Le passage de paramètres
11.3. VARIABLES PUBLIQUES ET PRIVEES
CHAPITRE 12 NOTIONS COMPLEMENTAIRES..41
12.1 INTERPRETATION ET COMPILATION
12.2. UNE LOGIQUE VICELARDE : LA PROGRAMMATION RECURSIVE

CHAPITRE 1
LE CODAGE

1.1. LES ORDINATEURS SONT « BINAIRES »

Un ordinateur manipule exclusivement des **informations binaires.**

Une information binaire est une information qui ne peut avoir que deux états : par exemple, ouvert - fermé, libre – occupé, blanc – noir, vrai – faux, etc.

On symbolise une information binaire, quel que soit son support physique, sous la forme de 1 et de 0. Il faut bien comprendre que ce n'est là qu'une **représentation**, que l'on utilise pour parler de toute information binaire.

1.2. LA NUMEROTATION DE POSITION EN BASE DECIMALE

L'homme utilise un système de **numérotation de position**, à **base décimale** pour représenter les nombres.

Pour représenter un nombre, nous disposons d'un alphabet spécialisé : une série de 10 signes qui s'appellent les chiffres. Et lorsque nous écrivons un nombre en mettant certains de ces chiffres les uns derrière les autres, l'ordre dans lequel nous mettons les chiffres est capital. Ainsi, par exemple, 2 569 n'est pas du tout le même nombre que 9 562. Et pourquoi ? Quel opération, quel décodage mental effectuons-nous lorsque nous lisons une suite de chiffres représentant un nombre ? Le problème, c'est que nous sommes tellement habitués à faire ce décodage de façon instinctive que généralement nous n'en connaissons plus les règles. Mais ce n'est pas très compliqué de les reconstituer… Et c'est là que nous mettons le doigt en plein dans la deuxième caractéristique de notre système de notation numérique : son caractère décimal.

On écrit que :

$9\,562 = 9 \times 10^3 + 5 \times 10^2 + 6 \times 10^1 + 2 \times 10^0$

C'est le mécanisme général de la représentation par numérotation de position en base décimale.

1.3. LA NUMEROTATION DE POSITION EN BASE BINAIRE

Les ordinateurs ont un dispositif physique fait pour stocker (de multiples façons) des informations binaires. Alors, lorsqu'on représente une information stockée par un ordinateur, le plus simple est d'utiliser un système de représentation à deux chiffres : les fameux 0 et 1.

Dans un ordinateur, le dispositif qui permet de stocker de l'information est donc rudimentaire, bien plus rudimentaire que les mains humaines. Avec des mains humaines, on peut coder dix choses différentes. Avec un emplacement d'information d'ordinateur, on est limité à deux choses différentes seulement. Avec une telle information binaire, on ne va pas loin. Voilà pourquoi, dès leur invention, les ordinateurs ont été conçus pour manier ces informations par paquets de 0 et de 1. Et la taille de ces paquets a été fixée à 8 informations binaires.

Une information binaire (symbolisée couramment par 0 ou 1) s'appelle un bit (en anglais... bit). Un groupe de huit bits s'appelle un octet (en anglais, byte) Donc, méfiance avec le byte (en abrégé, B majuscule), qui vaut un octet, c'est-à-dire huit bits (en abrégé, b minsucule).

Dans combien d'états différents un octet peut-il se trouver ? Le calcul est assez facile. Chaque bit de l'octet peut occuper deux états. Il y a donc dans un octet :

$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256$ possibilités

Cela signifie qu'un octet peut servir à coder 256 nombres différents : ce peut être la série des nombres entiers de 1 à 256, ou de 0 à 255, ou de –127 à +128. C'est une pure affaire de convention, de choix de codage. Mais ce qui n'est pas affaire de choix, c'est le nombre de possibilités : elles sont 256, pas une de plus, pas une de moins, à cause de ce qu'est, par définition, un octet.

Si l'on veut coder des nombres plus grands que 256, ou des nombres négatifs, ou des nombres décimaux, on va donc être contraint de mobiliser plus d'un octet. Ce n'est pas un problème, et c'est très souvent que les ordinateurs procèdent ainsi.

En effet, avec deux octets, on a 256 x 256 = 65 536 possibilités.

En utilisant trois octets, on passe à 256 x 256 x 256 = 16 777 216 possibilités.

Cela implique également qu'un octet peut servir à coder autre chose qu'un nombre : l'octet est très souvent employé pour coder du texte. Il y a 26 lettres dans l'alphabet. Même en comptant différemment les minuscules et les majuscules, et même en y ajoutant les chiffres et les signes de ponctuation, on arrive à un total inférieur à 256. Cela veut dire que pour coder convenablement un texte, le choix d'un caractère par octet est un choix pertinent.

Se pose alors le problème de savoir quel caractère doit être représenté par quel état de l'octet. Si ce choix était librement laissé à chaque informaticien, ou à chaque fabricant d'ordinateur, la communication entre deux ordinateurs serait impossible. Aussi, il existe un standard international de codage des caractères et des signes de ponctuation. Ce standard stipule quel état de l'octet correspond à quel signe du clavier. Il s'appelle l'ASCII (pour *American Standard Code for Information Interchange*). Et fort heureusement, l'ASCII est un standard universellement reconnu et appliqué par les fabricants d'ordinateurs et de logiciels. Bien sûr, se pose le problème des signes propres à telle ou telle langue (comme les lettres accentuées en français, par exemple). L'ASCII a paré le problème en réservant certains codes d'octets pour ces caractères spéciaux à chaque langue. En ce qui concerne les langues utilisant un alphabet non latin, un standard particulier de codage a été mis au point.

Nous avons vu qu'un octet pouvait coder 256 nombres différents, par exemple (c'est le choix le plus spontané) la série des entiers de 0 à 255. Comment faire pour, à partir d'un octet, reconstituer le nombre dans la base décimale qui nous est plus familière ? Ce n'est pas sorcier ; il suffit d'appliquer, si on les a bien compris, les principes de la numérotation de position, en gardant à l'esprit que là, la base n'est pas décimale, mais binaire. Prenons un octet au hasard :

1 1 0 1 0 0 1 1

D'après les principes vus plus haut, ce nombre représente en base dix, en partant de la gauche :

$1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 =$

$1 \times 128 + 1 \times 64 + 1 \times 16 + 1 \times 2 + 1 \times 1 =$

$128 + 64 + 16 + 2 + 1 =$

211

Inversement, comment traduire un nombre décimal en codage binaire ? Il suffit de rechercher dans notre nombre les puissances successives de deux. Prenons, par exemple, 186.

Dans 186, on trouve 1 x 128, soit 1×2^7. Je retranche 128 de 186 et j'obtiens 58.

Dans 58, on trouve 0 x 64, soit 0×2^6. Je ne retranche donc rien.

Dans 58, on trouve 1 x 32, soit 1 x 2^5. Je retranche 32 de 58 et j'obtiens 26.

Dans 26, on trouve 1 x 16, soit 1 x 2^4. Je retranche 16 de 26 et j'obtiens 10.

Dans 10, on trouve 1 x 8, soit 1 x 2^3. Je retranche 8 de 10 et j'obtiens 2.

Dans 2, on trouve 0 x 4, soit 0 x 2^2. Je ne retranche donc rien.

Dans 2, on trouve 1 x 2, soit 1 x 2^1. Je retranche 2 de 2 et j'obtiens 0.

Dans 0, on trouve 0 x 1, soit 0 x 2^0. Je ne retranche donc rien.

Il ne me reste plus qu'à reporter ces différents résultats (dans l'ordre !) pour reconstituer l'octet. J'écris alors qu'en binaire, 186 est représenté par :

1 0 1 1 1 0 1 0

C'est bon ? Alors on passe à la suite.

1.4. LE CODAGE HEXADECIMAL

Avec 4 bits, nous pouvons coder 2 x 2 x 2 x 2 = 16 nombres différents. En base seize, 16 nombres différents se représentent avec un seul chiffre (de même qu'en base 10, dix nombres se représentent avec un seul chiffre).

Les dix premiers nombres de la base seize s'écrivent 0, 1, 2, 3, 4, 5, 6, 7, 8, et 9. Là, il nous manque encore 6 chiffres, pour représenter les nombres que nous écrivons en décimal 10, 11, 12, 13, 14, 15 et 16. Plutôt qu'inventer de nouveaux symboles, on a utilisé les premières lettres de l'alphabet. Ainsi, par convention, A vaut 10, B vaut 11, etc. jusqu'à F qui vaut 15.

Or, on s'aperçoit que cette base hexadécimale permet une représentation très simple des octets du binaire. Prenons un octet au hasard :

1 0 0 1 1 1 1 0

Pour convertir ce nombre en hexadécimal, il y a deux méthodes : l'une consiste à faire un grand détour, en repassant par la base décimale. C'est un peu plus long, mais on y arrive. L'autre méthode consiste à faire le voyage direct du binaire vers l'hexadécimal.

Première méthode :

On retombe sur un raisonnement déjà abordé. Cet octet représente en base dix :

$1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 =$

1 x 128 + 1 x 16 + 1 x 8 + 1 x 4 + 1 x 2 + 0 x 1 =

128 + 16 + 8 + 4 + 2 =

158

De là, il faut repartir vers la base hexadécimale.

Dans 158, on trouve 9 x 16, c'est-à-dire 9 x 16^1. Je retranche 144 de 158 et j'obtiens 14.

Dans 14, on trouve 14 x 1, c'est-à-dire 14 x 16^0. On y est.

Le nombre s'écrit donc en hexadécimal : 9E

Deuxième méthode :

Divisons 1 0 0 1 1 1 1 0 en 1 0 0 1 (partie gauche) et 1 1 1 0 (partie droite).

1 0 0 1, c'est 8 + 1, donc 9

1 1 1 0, c'est 8 + 4 + 2 donc 14

Le nombre s'écrit donc en hexadécimal : 9E. C'est la même conclusion qu'avec la première méthode. Encore heureux !

Le codage hexadécimal est très souvent utilisé quand on a besoin de représenter les octets individuellement, car dans ce codage, tout octet correspond à seulement deux signes.

1.5. Conversion des entiers positifs

Binaire vers décimal

$1011001_2 = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
$= 64 + 0 + 16 + 8 + 0 + 0 + 1 = 89_{10}$

Décimal vers binaire

57 = 2 × 28+**1**
28 = 2 × 14+**0**
14 = 2 × 7+**0**
7 = 2 × 3+**1**
3 = 2 × **1**+**1**
$57_{10} = 111001_2$

Binaire vers hexadécimal
Le passage de binaire en hexadécimal se fait naturellement en divisant le nombre binaire par groupes de 4 chiffres, en partant de la fin du nombre :

$11011010101 = 110\ 1101\ 0101_2 = 6D5_{16}$

En effet, $110_2 = 6_{10} = 6_{16}$, $1101_2 = 13_{10} = D_{16}$, $0101_2 = 5_{10} = 5_{16}$

Conversion de n'importe quelle base vers la base 10

Il est utile de retenir le cas général, même si les calculs sont parfois peu pratiques. Soit un nombre de n chiffres écrit en base a : $\alpha_n \ldots \alpha_1$. Sa valeur décimale est $\Sigma^n{}_{i=1}\ \alpha_i \times a^{i-1}$ (le chiffre α_i, s'il n'est pas décimal, doit être remplacé par son équivalent décimal).

Conversion de la base 10 vers n'importe quelle base

Pour convertir un nombre décimal en base a, on effectue des divisions successives par a, jusqu'à ce que le dividende soit strictement plus petit que a. La suite des restes et le dernier dividende constitue la suite des chiffres du nombre en base a.

Par exemple, pour convertir 1423 en base 16 :

1423 ÷ 16 = 88, reste 15 (F en hexadécimal)
88 ÷ 16 = 5, reste 8

On a donc : 142310 = 58F16.

1.6. Arithmétique binaire

Les opérations d'addition et de multiplication se posent de la même manière qu'en décimal. Un peu de pratique permet de les effectuer rapidement.

Addition
Les tables d'additions binaires sont très simples : 0 + 0 = 0, 0 + 1 = 1 + 0 = 1, et 1 + 1 = 10 (retenue de 1). Voici un exemple d'addition posée :

```
   1 1 1
    101110
  +  11100
  ----------
= 1001010
```

Multiplication
La multiplication en binaire est tout à fait similaire à son équivalent décimal :

```
      101110
    ×  10100
    ----------
      000000
     000000
    101110
   000000
  101110
  --------------
   1  1  1
=1110011000
```

1.7. Mémoire de l'ordinateur et codage

La mémoire de l'ordinateur est un ensemble d'emplacements numérotés. Nous considèrerons que chaque emplacement peut contenir huit chiffres binaires (un octet). La taille de la mémoire d'un ordinateur est mesurée en octets ou en un de ses multiples : 1 kilo octet (Ko)=1024 octets, 1 méga octet (Mo)=1024 Ko, 1 giga octet (Go)=1024 Mo, 1 terra octet (To)=1024 Go. Typiquement de nos jours, un micro-ordinateur dispose de 256 Mo de mémoire (soit plus de 268 millions d'octets).
Dans ces cases sont stockées les données utilisées par la machine, que ce soient des caractères, de petits nombres, de grands nombres, des nombres négatifs... Il y a donc une méthode de codage qui indique de quelle façon, en utilisant uniquement des nombres binaires de 8 bits, on peut stocker de telles valeurs.

1.7.1 Nombres entiers positifs

Les petits nombres entiers positifs (compris entre 0 et 255) sont stockés sur un octet exactement (puisque $11111111_2 = 255_{10}$). Les nombres entiers plus grands sont stockés sur deux octets contigus. On peut alors atteindre des valeurs de : $11111111\ 11111111_2 = 65535_{10}$. On peut naturellement stocker des nombres encore plus grands en utilisant quatre octets contigus.

1.7.2. Codage des entiers relatifs en complément à deux

Un nombre positif est codé en base 2 comme détaillé précédemment. Un nombre négatif est codé de la façon suivante :
- coder la valeur absolue du nombre ;
- inverser tous les bits ;
- ajouter 1.

Du fait de la complémentation, il est nécessaire de savoir sur combien de chiffres binaires les nombres sont codés. Dans la suite on supposera que nous travaillons avec des nombres de 8 chiffres binaires.

Codage de 11410

$114_{10} = 1 \times 64 + 1 \times 32 + 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$
$114_{10} = 01110010_2$

Codage de -114_{10}

$114_{10} = 1 \times 64 + 1 \times 32 + 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$
$114_{10} = 01110010_2$
−11410 = 100011012 + 12 = 100011102

En travaillant sur 8 bits, les nombres binaires de 0 à 1111111 représentent les entiers positifs de 0 à 127 et les nombres binaires de 10000000 à 11111111 représentent les entiers négatifs de -128 à -1.

1.7.3. Codage des caractères

Pour coder les caractères, une table de correspondances, qui à un caractère fait correspondre un nombre est nécessaire. Il existe plusieurs de ces tables, la plus utilisée étant la table Ascii (American Standard Code for Information Interchange).
Il n'est pas du tout nécessaire de la connaître, mais il est très utile de savoir qu'elle existe. Il est important de comprendre que les caractères eux mêmes ne sont jamais inscrits en mémoire. Ce sont les nombres qui les codent qui sont inscrits.

CHAPITRE 2
INTRODUCTION A L'ALGORITHMIQUE

2.1. Algorithme

Un algorithme, c'est une suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné.

Pour fonctionner, **un algorithme doit contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter.**

Un algorithme très ancien et très connu est l'algorithme d'Euclide, qui permet de calculer le pgdc (plus grand diviseur commun) de deux entiers. On peut expliquer l'algorithme d'Euclide ainsi :
Etant donnés deux entiers, retrancher le plus petit au plus grand et recommencer jusqu'a ce que les deux nombres soient égaux. La valeur obtenue est le pgdc.
Voici un exemple : 144, 96 → 48, 96 → 48, 48 → Le pgdc de 144 et 96 est 48.

Pour être compris par une machine, un algorithme doit être écrit dans un langage spécifique, qu'on appelle langage de programmation. Un langage de programmation n'étant qu'une façon d'exprimer un algorithme, on comprend aisément que la notion d'algorithme doit être maîtrisée avant d'essayer d'écrire un programme dans un certain langage.

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.
- il faut être **méthodique** et **rigoureux**. En effet, 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.

Les ordinateurs, comprennent quatre catégories d'ordres (en programmation, on emploiera le terme **instructions**) qui sont :

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

Un algorithme informatique se ramène donc toujours 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.

2.2. ALGORITHMIQUE ET PROGRAMMATION

Pourquoi apprendre l'algorithmique pour apprendre à programmer ? En quoi a-t-on besoin d'un langage spécial, distinct des langages de programmation compréhensibles par les ordinateurs ?

Parce que l'algorithmique exprime les instructions résolvant un problème donné **indépendamment des particularités de tel ou tel langage**.

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.

2.3. AVEC QUELLES CONVENTIONS ECRIT-ON UN ALGORITHME ?

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 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 du tout. Ensuite parce que cette représentation favorise le glissement vers un certain type de programmation, dite non structurée (nous définirons ce terme plus tard), que l'on tente au contraire d'éviter.

C'est pourquoi on utilise généralement une série de conventions appelée « **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.

CHAPITRE 3
LES VARIABLES

3.1. A QUOI SERVENT LES VARIABLES ?

Dans un programme informatique, on a besoin de stocker des données issues du disque dur, fournies par l'utilisateur (frappées au clavier), ou de résultats obtenus par le programme. Ces données peuvent être de plusieurs types : des nombres, du texte, etc. Pour cela on utilise une **variable**. Une variable est une **boîte**, que le programme va repérer par une **étiquette**. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette.

Dans l'ordinateur, physiquement, il y a un emplacement de mémoire, repéré par une adresse binaire. Les langages informatiques plus évolués (ce sont ceux que presque tout le monde emploie) se chargent précisément, entre autres rôles, d'épargner au programmeur la gestion fastidieuse des emplacements mémoire et de leurs adresses. Et, comme vous commencez à le comprendre, il est beaucoup plus facile d'employer les étiquettes de son choix, que de devoir manier des adresses binaires.

3.2. DECLARATION DES VARIABLES

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**.

Un nom de variable peut comporter des lettres et des chiffres sans les espaces et commence par une lettre. Quant au nombre maximal de signes pour un nom de variable, il dépend du langage utilisé. En algorithmique, on est libre du nombre de signes pour un nom de variable, mais on évite généralement les noms trop longs.

Lorsqu'on déclare une variable, on doit préciser le **type de codage** utilisé.

3.2.1 Types numériques classiques

Commençons par le cas très fréquent, celui d'une variable destinée à recevoir des nombres.

Si l'on réserve un octet pour coder un nombre, on ne pourra coder que 2^8 = 256 valeurs différentes. Cela peut signifier par exemple les nombres entiers de 1 à 256, ou de 0 à 255, ou de -127 à +128… Si l'on réserve deux octets, on a droit à 65 536 valeurs ; avec trois octets, 16 777 216, etc. On retrouve en général les types numériques suivants avec les plages correspondantes:

Type Numérique	Plage
Byte (octet)	0 à 255
Entier simple	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647
Réel simple	$-3,40 \times 10^{38}$ à $-1,40 \times 10^{45}$ pour les valeurs négatives $1,40 \times 10^{-45}$ à $3,40 \times 10^{38}$ pour les valeurs positives
Réel double	$1,79 \times 10^{308}$ à $-4,94 \times 10^{-324}$ pour les valeurs négatives $4,94 \times 10^{-324}$ à $1,79 \times 10^{308}$ pour les valeurs positives

Une déclaration de variables aura ainsi cette forme :

```
Type Variable ;
Exemple Entier i ;
```

3.2.2 Type caractère

Car c ;

Dans une variable de ce type, on stocke des **caractères**, qu'il s'agisse de lettres, de signes de ponctuation, d'espaces, ou même de chiffres. Le nombre maximal de caractères pouvant être stockés dans une seule variable caractère dépend du langage utilisé.

Un groupe de caractères (y compris un groupe de un, ou de zéro caractères), est appelé **chaîne** de caractères.

En algorithme, une chaîne de caractères est toujours notée entre guillemets

3.2.3 Type booléen

On y stocke uniquement les valeurs logiques VRAI et FAUX.

Logique l ;

3.3. L'INSTRUCTION D'AFFECTATION

3.3.1 Syntaxe et signification

La première chose qu'on puisse faire avec une variable, c'est **lui affecter une valeur,** c'est-à-dire **lui attribuer une valeur**.

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

exemple :

```
Toto ← 24 ;
```

Attribue la valeur 24 à la variable Toto.

On peut attribuer à une variable la valeur d'une autre variable, par exemple :

```
Tutu ← Toto ;
```

Signifie que la valeur de Tutu est maintenant celle de Toto.

une instruction d'affectation ne modifie que ce qui est situé à gauche de la flèche.

```
Tutu ← Toto + 4 ;
```

Si Toto contenait 12, Tutu vaut maintenant 16. Toto vaut toujours 12.

```
Tutu ← Tutu + 1 ;
```

Si Tutu valait 6, il vaut maintenant 7. La valeur de Tutu est modifiée, puisque Tutu est la variable située à gauche de la flèche.

3.3.2 Ordre des instructions

Il va de soi que l'ordre dans lequel les instructions sont écrites va jouer un rôle essentiel dans le résultat final. Considérons les deux algorithmes suivants :

```
Exemple 1
Début
Réel A ;
A ← 34 ;
A ← 12 ;
Fin
```

```
Exemple 2
Début
Réel A ;
A ← 12 ;
A ← 34 ;
Fin
```

Il est clair que dans le premier cas la valeur finale de A est 12, dans l'autre elle est 34 .

3.4. EXPRESSIONS ET OPERATEURS

Si on fait le point, on s'aperçoit que dans une instruction d'affectation, on trouve :

- à gauche de la flèche, un nom de variable, et uniquement cela.
- à droite de la flèche, ce qu'on appelle une **expression**.

Une expression est un ensemble de valeurs, reliées par des opérateurs, et équivalent à une seule valeur

Un opérateur est un signe qui relie deux valeurs, pour produire un résultat.

Les opérateurs possibles dépendent du type des valeurs qui sont en jeu.

3.4.1 Opérateurs numériques :

Ce sont les quatre opérations arithmétiques :

+ : addition

- : soustraction

* : multiplication

/ : division

Mentionnons également le ^ qui signifie « puissance ». 45 au carré s'écrira donc 45 ^ 2.

Enfin, on a le droit d'utiliser les parenthèses, avec les mêmes règles qu'en mathématiques. La multiplication et la division ont « naturellement » priorité sur l'addition et la soustraction. Les parenthèses ne sont ainsi utiles que pour modifier cette priorité naturelle.

Cela signifie qu'en informatique, 12 * 3 + 5 et (12 * 3) + 5 valent strictement la même chose, à savoir 41. En revanche, 12 * (3 + 5) vaut 12 * 8 soit 96.

3.4.2 Opérateur alphanumérique : &

Cet opérateur permet de **concaténer**, autrement dit d'agglomérer, deux chaînes de caractères. Par exemple :

```
Début
car A, B, C ;
A ← "Alger"
B ← "ie"
C ← A & B
Fin
```

La valeur de C à la fin de l'algorithme est "Algerie"

3.4.3 Opérateurs logiques (ou booléens) : Il s'agit du ET, du OU, et du NON.

CHAPITRE 4
LECTURE ET ECRITURE

4.1. INTRODUCTION

Soit le programme suivant pour calculer le carré du nombre 12 :

```
Réel A ;
A ← 12**2
```

Ce programme nous donne le carré de 12. Mais si l'on veut le carré d'un autre nombre que 12, il faut réécrire le programme.

D'autre part, le résultat est calculé par la machine. Mais elle le garde soigneusement pour elle, et l'utilisateur qui fait exécuter ce programme, lui, ne saura jamais quel est le carré de 12.

C'est pourquoi, il existe des instructions pour permettre à la machine de dialoguer avec l'utilisateur. Ces instructions permettent à l'utilisateur de rentrer des valeurs au clavier pour qu'elles soient utilisées par le programme. Cette opération est la **lecture**.

Dans l'autre sens, d'autres instructions permettent au programme de communiquer des valeurs à l'utilisateur en les affichant à l'écran. Cette opération est **l'écriture**.

4.2. LES INSTRUCTIONS DE LECTURE ET D'ECRITURE

```
Lire Titi ;
```

Dès que le programme rencontre une instruction Lire, l'exécution s'interrompt, attendant la frappe d'une valeur au clavier

Dès lors, aussitôt que la touche Entrée (Enter) a été frappée, l'exécution reprend. Dans le sens inverse, pour écrire quelque chose à l'écran, c'est aussi simple que :

```
Ecrire Toto ;
```

Avant de Lire une variable, il est très fortement conseillé d'écrire des **libellés** à l'écran, afin de prévenir l'utilisateur de ce qu'il doit frapper :

```
Ecrire ("Entrez votre nom : ") ;
Lire NomFamille ;
```

CHAPITRE 5
LES TESTS

5.1. DEFINITION

Un test c'est prévoir, en fonction d'une situation pouvant se présenter de deux façons différentes, deux façons différentes d'agir.

5.2. STRUCTURE D'UN TEST

Il y a **deux formes possibles** pour un test :

```
Si condition
 Alors Instructions
Finsi

Si condition
 Alors Instructions 1
Sinon  Instructions 2
Finsi
```

Une condition est une **expression** dont la valeur est VRAI ou FAUX.

Dans la forme la plus simple, arrivé à ligne (Si… Alors) la machine examine la valeur de la condition, si elle a pour valeur VRAI, elle exécute la série d'instructions. Cette série d'instructions peut être très brève comme très longue, cela n'a aucune importance. En revanche, dans le cas où la condition est fausse, l'ordinateur saute directement aux instructions situées après le FinSi.

Dans le cas de la structure complète, et dans le cas où la condition est VRAI, et après avoir exécuté la série d'instructions 1, au moment où elle arrive au mot « Sinon », la machine saute directement à la première instruction située après le « Finsi ». De même, au cas où la condition a comme valeur « Faux », la machine saute directement à la première ligne située après le « Sinon » et exécute l'ensemble des « instructions 2 ». Dans tous les cas, les instructions situées juste après le FinSi seront exécutées normalement.

5.3. QU'EST CE QU'UNE CONDITION ?

Une condition est une comparaison

Elle signifie qu'une condition est composée de trois éléments :

- une valeur
- un **opérateur de comparaison**
- une autre valeur

Les valeurs peuvent être a priori de n'importe quel type (numériques, caractères…). Mais si l'on veut que la comparaison ait un sens, il faut que les deux valeurs de la comparaison soient du même type.

Les **opérateurs de comparaison** sont : **= <> < <= > >=**

L'ensemble des trois éléments constituant la condition constitue donc, si l'on veut, une affirmation, qui a un moment donné est VRAIE ou FAUSSE.

5.4. CONDITIONS COMPOSEES

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci-dessus. Reprenons le cas « Toto est inclus entre 5 et 8 ». En fait cette phrase cache non une, mais **deux** conditions. Car elle revient à dire que « Toto est supérieur à 5 et Toto est inférieur à 8 ». Il y a donc bien là deux conditions, reliées par ce qu'on appelle un **opérateur logique**, le mot ET.

Nous avons à notre disposition quatre opérateurs logiques : ET, OU, NON, et XOR.

- Le ET a le même sens en informatique que dans le langage courant. Pour que "Condition1 ET Condition2" soit VRAI, il faut impérativement que Condition1 soit VRAI et que Condition2 soit VRAI. Dans tous les autres cas, "Condition 1 et Condition2" sera faux.
- Pour que "Condition1 OU Condition2" soit VRAI, il suffit que Condition1 soit VRAIE ou que Condition2 soit VRAIE. Le point important est que si Condition1 est VRAIE et que Condition2 est VRAIE aussi, Condition1 OU Condition2 reste VRAIE.
- Le XOR (ou OU exclusif) fonctionne de la manière suivante. Pour que "Condition1 XOR Condition2" soit VRAI, il faut que soit Condition1 soit VRAI, soit que Condition2 soit VRAI. Si toutes les deux sont fausses, ou que toutes les deux sont VRAI, alors le résultat global est considéré comme FAUX.
- Le NON(Condition1)est VRAI si Condition1 est FAUX, et il sera FAUX si Condition1 est VRAI.

Dans une condition composée employant à la fois des opérateurs ET et des opérateurs OU, la présence de parenthèses possède une influence sur le résultat, tout comme dans le cas d'une expression numérique comportant des multiplications et des additions.

5.5. TESTS IMBRIQUES

Il y a des situations où nous avons besoin de plus de deux états pour prendre une décision. Par exemple, un programme devant donner l'état de l'eau selon sa température doit pouvoir choisir entre trois réponses possibles (solide, liquide ou gazeuse).

Une première solution serait la suivante :

```
Début
Entier Temp ;
Ecrire ("Entrez la température de l'eau :") ;
Lire Temp ;
Si Temp <= 0 Alors
  Ecrire ("C'est de la glace") ;
FinSi
Si (Temp > 0) Et (Temp < 100) Alors
  Ecrire ("C'est du liquide") ;
Finsi
Si Temp > 100 Alors
  Ecrire ("C'est de la vapeur") ;
Finsi
Fin
```

Les conditions se ressemblent plus ou moins, et on oblige la machine à examiner trois tests successifs alors que tous portent sur une même chose, la température de l'eau (la valeur de la variable Temp). Il serait ainsi bien plus rationnel d'**imbriquer** les tests de cette manière :

```
Début
Entier Temp ;
Ecrire ("Entrez la température de l'eau :") ;
Lire Temp ;
Si Temp <= 0
  Alors
        Ecrire ("C'est de la glace") ;
  Sinon
       Si (Temp < 100)
       Alors
             Ecrire ("C'est du liquide") ;
       Sinon
             Ecrire ("C'est de la vapeur") ;
       Finsi
Finsi
Fin
```

Cette deuxième version n'est pas seulement plus simple à écrire et plus lisible, elle est également plus performante à l'exécution.

CHAPITRE 6
LES BOUCLES

Appelées aussi **structures répétitives**, ou **structures itératives**.

On peut dire que les boucles constituent la seule vraie structure logique caractéristique de la programmation.

6.1. LA BOUCLE TANTQUE

Une telle **structure de boucle**, se présente ainsi :

```
TantQue condition
Faire    Instructions
FinTantQue
```

Le principe est simple : le programme arrive sur la ligne du TantQue. Il examine alors la valeur de la condition. Si cette valeur est VRAI, le programme exécute les instructions qui suivent, jusqu'à ce qu'il rencontre la ligne FinTantQue. Il retourne ensuite sur la ligne du TantQue, procède au même examen, et ainsi de suite. La répétition ne s'arrête que lorsque la condition prend la valeur FAUX.

6.2. LA BOUCLE POUR

Il arrive très souvent qu'on ait besoin d'effectuer un nombre **déterminé** de passages. Or, a priori, notre structure TantQue ne sait pas à l'avance combien de tours de boucle elle va effectuer.

C'est pourquoi une autre structure de boucle est à notre disposition :

```
Pour compteur de ValInitiale à Valfinale pas Valpas
 Faire Instructions
FinPour
```

La structure « Pour … Faire … » n'est pas du tout indispensable ; on pourrait fort bien programmer toutes les situations de boucle uniquement avec un « Tant Que ». Le seul intérêt du « Pour » est d'épargner un peu de fatigue au programmeur, en lui évitant de gérer lui-même la progression de la variable qui lui sert de compteur (on parle d'**incrémentation**).

La structure « Pour …Faire… » est un cas particulier de TantQue : celui où le programmeur peut dénombrer à l'avance le nombre de tours de boucles nécessaires.

Il faut noter que dans une structure « Pour … Faire… », la progression du compteur est laissée à votre libre disposition. Dans la plupart des cas, on a besoin d'une variable qui augmente de 1 à chaque tour de boucle. On ne précise alors rien à l'instruction « Pour » ; celle-ci, par défaut, comprend qu'il va falloir procéder à cette incrémentation de 1 à chaque passage, en commençant par la première valeur ValInitiale et en terminant par la valeur Valfinale.

Mais si vous souhaitez une progression plus spéciale, de 2 en 2, ou de 3 en 3, ou en arrière, de -1 en -1, ou de -10 en -10, ce n'est pas un problème : il suffira de le préciser à votre instruction « Pour » en lui rajoutant le mot « Pas » et la valeur de ce pas Valpas.

Naturellement, quand on stipule un pas négatif dans une boucle, la valeur initiale du compteur doit être **supérieure** à sa valeur finale si l'on veut que la boucle tourne ! Dans le cas contraire, on aura simplement écrit une boucle dans laquelle le programme ne rentrera jamais.

Les structures **TantQue** sont employées dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'un ensemble dont on ne connaît pas d'avance la quantité.

Les structures **Pour** sont employées dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'un ensemble dont le programmeur connaît d'avance la quantité.

6.3. LES BOUCLES IMBRIQUEES

De même que les poupées russes contiennent d'autres poupées russes, de même qu'une structure SI … ALORS peut contenir d'autres structures SI … ALORS, une boucle peut tout à fait contenir d'autres boucles.

```
Pour compteur1 de ValInitiale1 à Valfinale1 pas Valpas1
 Faire Pour compteur2 de ValInitiale2 à Valfinale2 pas Valpas2
          Faire Instructions
       FinPour
  FinPour
```

```
TantQue condition1
Faire   TantQue condition2
          Faire  Instructions
        FinTantQue
FinTantQue
```

Une boucle est un traitement systématique, un examen d'une série d'éléments un par un (par exemple, « prenons tous les employés de l'entreprise un par un »). Eh bien, on peut imaginer que pour chaque élément ainsi considéré (pour chaque employé), on doive procéder à un examen systématique d'autre chose (« prenons chacune des commandes que cet employé a traitées »). Voilà un exemple typique de boucles imbriquées : on devra programmer une boucle principale (celle qui prend les employés un par un) et à l'intérieur, une boucle secondaire (celle qui prend les commandes de cet employé une par une).

CHAPITRE 7
LES TABLEAUX

7.1. UTILITE DES TABLEAUX

Imaginons que dans un programme, nous ayons besoin simultanément de 12 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer douze variables, appelées par exemple Notea, Noteb, Notec, etc. Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc. Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, et après une succession de douze instructions « Lire » distinctes, cela donnera obligatoirement une écriture du genre :

```
Moy ← (N1+N2+N3+N4+N5+N6+N7+N8+N9+N10+N11+N12)/12 ;
```

C'est tout de même laborieux. Et pour un peu que nous soyons dans un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter, alors là c'est la catastrophe.

C'est pourquoi la programmation nous permet **de rassembler toutes ces variables en une seule**, au sein de laquelle chaque valeur sera désignée par un numéro.

Un ensemble de valeurs portant le même nom de variable et repérées par un nombre, s'appelle un tableau, ou encore une variable indicée.
Le nombre qui, au sein d'un tableau, sert à repérer chaque valeur s'appelle l'indice.
Chaque fois que l'on doit désigner un élément du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément, entre parenthèses.

7.2. NOTATION ET UTILISATION ALGORITHMIQUE

Dans notre exemple, nous créerons donc un tableau appelé Note. Chaque note individuelle (chaque élément du tableau Note) sera donc désignée Note(1), Note(2), etc.

Un tableau doit être déclaré comme tel, en précisant le nombre et le type de valeurs qu'il contiendra (la déclaration des tableaux est susceptible de varier d'un langage à l'autre. Certains langages réclament le nombre d'éléments, d'autre le plus grand indice... C'est donc une affaire de conventions).

```
Tableau Entier Note(12) ;
```

On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, bien sûr, mais aussi tableaux de caractères, tableaux de booléens, tableaux de tout ce qui existe dans un langage donné comme type de variables. Par contre, on ne peut pas faire un mixage de types différents de valeurs au sein d'un même tableau.

L'énorme avantage des tableaux, c'est qu'on va pouvoir les traiter en faisant des boucles. Par exemple, pour effectuer notre calcul de moyenne, cela donnera par exemple :

```
Tableau Entier Note(12) ;
Réel Moy, Som
Pour i de 1 à 12
  Faire
       Ecrire ("Entrez la note n :", i) ;
       Lire Note(i) ;
  FinPour
Som ← 0 ;
```

```
Pour i de 1 à 12
  Faire
        Som ← Som + Note(i);
FinPour
Moy ← Som / 12;
```

NB : On a fait deux boucles successives pour plus de lisibilité, mais on aurait tout aussi bien pu n'en écrire qu'une seule dans laquelle on aurait tout fait d'un seul coup.

Remarque générale : l'indice qui sert à désigner les éléments d'un tableau peut être exprimé directement comme un nombre en clair, mais il peut être aussi une variable, ou une expression calculée.

Dans un tableau, la valeur d'un indice doit toujours :

- **être égale au moins à 1** (dans la plupart des langages, le premier élément d'un tableau porte l'indice 0).
- **être un nombre entier** Quel que soit le langage.
- **être inférieure ou égale au nombre d'éléments du tableau**.

7.3. TABLEAUX DYNAMIQUES

Il arrive fréquemment que l'on ne connaisse pas à l'avance le nombre d'éléments que devra comporter un tableau. Bien sûr, une solution consisterait à déclarer un tableau gigantesque (10 000 éléments !) pour être sûr que « ça rentre ». Mais d'une part, on n'en sera jamais parfaitement sûr, d'autre part, en raison de l'immensité de la place mémoire réservée – et la plupart du temps non utilisée, c'est un gâchis préjudiciable à la rapidité, voire à la viabilité, de notre algorithme.

Aussi, pour parer à ce genre de situation, a-t-on la possibilité de déclarer le tableau sans préciser au départ son nombre d'éléments. Ce n'est que dans un second temps, au cours du programme, que l'on va fixer ce nombre via une instruction de redimensionnement : **Redim**.

Un tel tableau est un tableau dynamique.

Notez que **tant qu'on n'a pas précisé le nombre d'éléments d'un tableau, d'une manière ou d'une autre, ce tableau est inutilisable**.

7.4. TABLEAUX A DEUX DIMENSIONS

L'informatique nous offre la possibilité de déclarer des tableaux dans lesquels les valeurs ne sont pas repérées par une seule, mais par **deux coordonnées**.

Un tel tableau se déclare ainsi :

```
Tableau Entier Cases(7, 7) ;
```

Cela veut dire : réserve moi un espace de mémoire pour 8 x 8 entiers, et quand j'aurai besoin de l'une de ces valeurs, je les repèrerai par deux indices (comme à la bataille navale, ou Excel, la seule différence étant que pour les coordonnées, on n'utilise pas de lettres, juste des chiffres).

Une case est écrite Cases(i, j).

Dans un tableau à deux dimensions, le premier indice représente les lignes et le deuxième les colonnes.

7.5. TABLEAUX A N DIMENSIONS

Si vous avez compris le principe des tableaux à deux dimensions, sur le fond, il n'y a aucun problème à passer au maniement de tableaux à trois, quatre, ou pourquoi pas neuf dimensions. C'est exactement la même chose. Si je déclare un tableau Titi(2, 4, 3, 3), il s'agit d'un espace mémoire contenant 3 x 5 x 4 x 4 = 240 valeurs. Chaque valeur y est repérée par quatre coordonnées.

Pour des raisons uniquement pratiques, les tableaux à plus de trois dimensions sont rarement utilisés par des programmeurs non matheux.

CHAPITRE 8
QUELQUES TECHNIQUES CELEBRES

8.1. TRI D'UN TABLEAU : LE TRI PAR SELECTION

On a souvent besoin de ranger des valeurs dans un ordre donné.

Il existe plusieurs stratégies possibles pour trier les éléments d'un tableau ; nous en verrons deux : le tri par sélection, et le tri à bulles.

Commençons par le tri par sélection.

Admettons que le but de la manœuvre soit de trier un tableau de 12 éléments dans l'ordre croissant. La technique du tri par sélection est la suivante : on met en bonne position l'élément numéro 1, c'est-à-dire le plus petit. Puis en met en bonne position l'élément suivant. Et ainsi de suite jusqu'au dernier. Par exemple, si l'on part de :

45	122	12	3	21	78	64	53	89	28	84	46

On commence par rechercher, parmi les 12 valeurs, quel est le plus petit élément, et où il se trouve. On l'identifie en quatrième position (c'est le nombre 3), et on l'échange alors avec le premier élément (le nombre 45). Le tableau devient ainsi :

3	122	12	**45**	21	78	64	53	89	28	84	46

On recommence à chercher le plus petit élément, mais cette fois, **seulement à partir du deuxième** (puisque le premier est maintenant correct, on n'y touche plus). On le trouve en troisième position (c'est le nombre 12). On échange donc le deuxième avec le troisième :

3	**12**	**122**	45	21	78	64	53	89	28	84	46

On recommence à chercher le plus petit élément à partir du troisième (puisque les deux premiers sont maintenant bien placés), et on le place correctement, en l'échangeant, ce qui donnera :

3	12	**21**	45	**122**	78	64	53	89	28	84	46

Ainsi de suite jusqu'à l'avant dernier.

nous pourrions décrire le processus de la manière suivante :

- Boucle principale : prenons comme point de départ le premier élément, puis le second, etc, jusqu'à l'avant dernier.

- Boucle secondaire : à partir de ce point de départ mouvant, recherchons jusqu'à la fin du tableau quel et le plus petit élément. Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ.

Cela s'écrit :

```
Pour i de 1 à 11
Faire
    posmini ← i ;
  Pour j de i + 1 à 12
  Faire Si t(j) < t(posmini)
           Alors posmini ← j ;
       Finsi
  FinPour
  temp ← t(posmini)
  t(posmini) ← t(i)
  t(i) ← temp
FinPour
```

8.2. UN EXEMPLE DE FLAG : LA RECHERCHE DANS UN TABLEAU

Nous allons maintenant nous intéresser au maniement habile d'une variable booléenne : la technique dite du « **flag** ».

Le flag, en anglais, est un petit drapeau, qui va rester baissé aussi longtemps que l'événement attendu ne se produit pas. Et, aussitôt que cet événement a lieu, le petit drapeau se lève (la variable booléenne change de valeur). Ainsi, la valeur finale de la variable booléenne permet au programmeur de savoir si l'événement a eu lieu ou non.

Cela devrait s'éclairer à l'aide d'un exemple : la recherche de l'occurrence d'une valeur dans un tableau.

Soit un tableau comportant 20 valeurs. L'on doit écrire un algorithme saisissant un nombre au clavier, et qui informe l'utilisateur de la présence ou de l'absence de la valeur saisie dans le tableau.

La première étape, évidente, consiste à écrire les instructions de lecture / écriture, et la boucle de parcours du tableau :

```
Tableau entier Tab(20) ;
Entier n ;
Ecrire ("Entrez la valeur à rechercher") ;
Lire n ;
Pour i ← 1 à 20
Faire ???
FinPour
Fin
```

Il nous reste à combler les points d'interrogation de la boucle Pour. Évidemment, il va falloir comparer n à chaque élément du tableau : si les deux valeurs sont égales, alors N fait partie du tableau. Cela va se traduire, bien entendu, par un Si … Alors … Sinon.

```
Tableau entier Tab(20) ;
Entier n ;
Ecrire ("Entrez la valeur à rechercher") ;
Lire n ;
Pour i ← 1 à 20
```

```
Faire ???
FinPour
Si Trouvé Alors
  Ecrire ("N fait partie du tableau") ;
Sinon
  Ecrire ("N ne fait pas partie du tableau") ;
FinSi
Fin
```

Il ne nous reste plus qu'à gérer la variable Trouvé. Ceci se fait en deux étapes.

- un test figurant dans la boucle, indiquant lorsque la variable Trouvé doit devenir vraie (à savoir, lorsque la valeur N est rencontrée dans le tableau).
- l'affectation par défaut de la variable Trouvé, dont la valeur de départ doit être évidemment Faux.

Au total, l'algorithme complet donne :

```
Tableau entier Tab(20) ;
Entier n ;
Ecrire ("Entrez la valeur à rechercher") ;
Lire n ;
Trouvé ← Faux
Pour i ← 1 à 20
Faire Si N = Tab(i)
       Alors Trouvé ← Vrai ;
      FinSi
FinPour
Si Trouvé Alors
  Ecrire ("N fait partie du tableau") ;
Sinon
  Ecrire ("N ne fait pas partie du tableau") ;
FinSi
Fin
```

il suffit que n soit égal à une seule valeur de Tab pour qu'elle fasse partie du tableau. En revanche, il faut qu'elle soit différente de toutes les valeurs de Tab pour qu'elle n'en fasse pas partie.

Voilà la raison qui nous oblige à passer par une variable booléenne. Et cette technique de flag doit être mise en œuvre chaque fois que l'on se trouve devant pareille situation.

8.3. TRI DE TABLEAU + FLAG = TRI A BULLES

L'idée de départ du tri à bulles consiste à se dire qu'un tableau trié en ordre croissant, c'est un tableau dans lequel **tout élément est plus petit que celui qui le suit**.

En effet, prenons chaque élément d'un tableau, et comparons-le avec l'élément qui le suit. Si l'ordre n'est pas bon, on permute ces deux éléments. Et on recommence jusqu'à ce que l'on n'ait plus aucune permutation à effectuer. Les éléments les plus grands « remontent » ainsi peu à peu vers les dernières places, ce qui explique la charmante dénomination de « tri à bulle ».

En quoi le tri à bulles implique-t-il l'utilisation d'un flag ? Eh bien, par ce qu'on ne sait jamais par avance combien de remontées de bulles on doit effectuer. En fait, tout ce qu'on peut dire, c'est qu'on devra effectuer le tri jusqu'à ce qu'il n'y ait plus d'éléments qui soient mal classés.

Ceci est typiquement un cas de question « asymétrique » : il suffit que deux éléments soient mal classés pour qu'un tableau ne soit pas trié. En revanche, il faut que tous les éléments soient bien rangés pour que le tableau soit trié.

Nous baptiserons le flag Yapermute, car cette variable booléenne va nous indiquer si nous venons ou non de procéder à une permutation au cours du dernier balayage du tableau (dans le cas contraire, c'est signe que le tableau est trié, et donc qu'on peut arrêter la machine à bulles). La boucle principale sera alors :

```
Booléen Yapermute ;
…
TantQue Yapermute
…
FinTantQue
Fin
```

Que va-t-on faire à l'intérieur de la boucle ? Prendre les éléments du tableau, du premier jusqu'à l'avant-dernier, et procéder à un échange si nécessaire :

```
Booléen Yapermute ;
…
TantQue Yapermute
Pour i de 1 à 11
 Faire Si t(i) > t(i+1)
       Alors  temp ← t(i)
              t(i) ← t(i+1)
              t(i+1) ← temp
       Finsi
FinPour
```

Mais il ne faut pas oublier un détail capital : la gestion de notre flag. L'idée, c'est que cette variable va nous signaler le fait qu'il y a eu au moins une permutation effectuée. Il faut donc :

- lui attribuer la valeur Vrai dès qu'une seule permutation a été faite (il suffit qu'il y en ait eu une seule pour qu'on doive tout recommencer encore une fois).
- la remettre à Faux à chaque tour de la boucle principale (quand on recommence un nouveau tour général de bulles, il n'y a pas encore eu d'éléments échangés),
- dernier point, il ne faut pas oublier de lancer la boucle principale, et pour cela de donner la valeur Vrai au flag au tout départ de l'algorithme.

La solution complète donne donc :

```
Booléen Yapermute ;
…
Yapermut ← Vrai ;
TantQue Yapermute
Faire Yapermut ← Faux ;
Pour i de 1 à 11
 Faire Si t(i) > t(i+1)
       Alors  temp ← t(i) ;
              t(i) ← t(i+1) ;
              t(i+1) ← temp ;
              Yapermut ← Vrai ;
       Finsi
FinPour
FinTanQue
```

8.4. LA RECHERCHE DICHOTOMIQUE

Imaginons que nous ayons un programme qui doive vérifier si un mot existe dans le dictionnaire. Nous pouvons supposer que le dictionnaire a été préalablement entré dans un tableau (à raison d'un mot par emplacement). Ceci peut nous mener à, disons à 40 000 mots.

Une première manière de vérifier si un mot se trouve dans le dictionnaire consiste à examiner successivement tous les mots du dictionnaire, du premier au dernier, et à les comparer avec le mot à vérifier. Ca marche, mais cela risque d'être long : si le mot ne se trouve pas dans le dictionnaire, le programme ne le saura qu'après 40 000 tours de boucle ! Et même si le mot figure dans le dictionnaire, la réponse exigera tout de même en moyenne 20 000 tours de boucle. C'est beaucoup, même pour un ordinateur.

Or, il y a une autre manière de chercher, bien plus intelligente pourrait-on dire, et qui met à profit le fait que dans un dictionnaire, les mots sont triés par ordre alphabétique. D'ailleurs, un être humain qui cherche un mot dans le dictionnaire ne lit jamais tous les mots, du premier au dernier : il utilise lui aussi le fait que les mots sont triés.

Pour une machine, quelle est la manière la plus rationnelle de chercher dans un dictionnaire ? C'est de comparer le mot à vérifier avec le mot qui se trouve au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans la première moitié du dictionnaire. Sinon, on sait maintenant qu'on devra le chercher dans la deuxième moitié.

A partir de là, on prend la moitié de dictionnaire qui nous reste, et on recommence : on compare le mot à chercher avec celui qui se trouve au milieu du morceau de dictionnaire restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite.

A force de couper notre dictionnaire en deux, puis encore en deux, etc. on va finir par se retrouver avec des morceaux qui ne contiennent plus qu'un seul mot. Et si on n'est pas tombé sur le bon mot à un moment ou à un autre, c'est que le mot à vérifier ne fait pas partie du dictionnaire.

Regardons ce que cela donne en terme de nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire.

- Au départ, on cherche le mot parmi 40 000.
- Après le test n°1, on ne le cherche plus que parmi 20 000.
- Après le test n°2, on ne le cherche plus que parmi 10 000.
- Après le test n°3, on ne le cherche plus que parmi 5 000.
- etc.
- Après le test n°15, on ne le cherche plus que parmi 2.
- Après le test n°16, on ne le cherche plus que parmi 1.

Et là, on sait que le mot n'existe pas. Moralité : on a obtenu notre réponse en 16 opérations contre 40 000 précédemment !

Au risque de me répéter, la compréhension et la maîtrise du principe du flag font partie du bagage du programmeur bien outillé.

CHAPITRE 9
LES FONCTIONS PREDEFINIES

Pour calculer le sinus d'un angle on doit obtenir une valeur approchée, et pour cela il faudrait appliquer une formule. Que se passe-t-il sur les petites calculatrices ? On fournit quelques touches spéciales, dites **touches de fonctions**, qui permettent par exemple de connaître immédiatement ce résultat. Si on veut connaître le sinus de 35°, on tape 35, puis la touche SIN, et nous avons le résultat.

Tout langage de programmation propose ainsi un certain nombre de **fonctions** ; certaines sont indispensables, car elles permettent d'effectuer des traitements qui seraient sans elles impossibles. D'autres servent à soulager le programmeur, en lui épargnant de longs – et pénibles - algorithmes.

9.1. STRUCTURE GENERALE DES FONCTIONS

Reprenons l'exemple de SIN. Les langages informatiques doivent savoir faire la même chose qu'une calculatrice ; ils proposent généralement une fonction SIN. Si nous voulons stocker le sinus de 35 dans la variable A, nous écrivons :

```
A ← Sin(35) ;
```

Une fonction est donc constituée de trois parties :

- le **nom** proprement dit de la fonction. Il doit impérativement correspondre à une fonction proposée par le langage. Dans notre exemple, ce nom est SIN.
- deux parenthèses, une ouvrante, une fermante. Ces parenthèses sont toujours **obligatoires**, même lorsqu'on n'écrit rien à l'intérieur.
- une liste de valeurs, indispensables à la bonne exécution de la fonction. Ces valeurs s'appellent des **arguments**, ou des **paramètres**. Certaines fonctions exigent un seul argument, d'autres deux, etc. et d'autres encore aucun. A noter que même dans le cas de ces fonctions n'exigeant aucun argument, les parenthèses restent obligatoires. Le nombre d'arguments nécessaire pour une fonction donnée est fixé par le langage. Par exemple, la fonction sinus a besoin d'un argument. Notons également que les arguments doivent être d'un certain **type**, et qu'il faut respecter ces types.

9.2. LES FONCTIONS DE TEXTE

Une catégorie de fonctions est celle qui nous permet de manipuler des chaînes de caractères. Nous avons déjà vu qu'on pouvait facilement « coller » deux chaînes l'une à l'autre avec l'opérateur de concaténation &. Mais ce que nous ne pouvions pas faire, et qui va être maintenant possible, c'est pratiquer des extractions de chaînes.

- **Len(chaîne)** : renvoie le nombre de caractères d'une chaîne
- **Mid(chaîne,n1,n2)** : renvoie un extrait de la chaîne, commençant au caractère n1 et faisant n2 caractères de long.

Ce sont les deux seules fonctions de chaînes réellement indispensables. Cependant, pour nous épargner des algorithmes fastidieux, les langages proposent également :

- **Left(chaîne,n)** : renvoie les n caractères les plus à gauche dans chaîne.
- **Right(chaîne,n)** : renvoie les n caractères les plus à droite dans chaîne
- **Trouve(chaîne1,chaîne2)** : renvoie un nombre correspondant à la position de chaîne2 dans chaîne1. Si chaîne2 n'est pas comprise dans chaîne1, la fonction renvoie zéro.

Il existe aussi dans tous les langages une fonction qui renvoie le caractère correspondant à un code Ascii donné (fonction **Asc**), et inversement (fonction **Chr**) :

```
Asc("N")                                vaut      78
Chr(63)                                 vaut      "?"
```

9.3. TROIS FONCTIONS NUMERIQUES CLASSIQUES

Partie Entière

Une fonction extrêmement répandue est celle qui permet de récupérer la partie entière d'un nombre :

```
Après :   A ← Ent(3.228) ;                 A vaut 3
```

Modulo

Cette fonction permet de récupérer le reste de la division d'un nombre par un deuxième nombre. Par exemple :

```
A ← Mod(10,3) ;                A vaut 1 car 10 = 3*3 + 1
```

Génération de nombres aléatoires

Une autre fonction classique, est celle qui génère un nombre choisi au hasard.

Dans tous les langages, cette fonction existe et produit le résultat suivant :

```
Après :   Toto ← Alea()            On a :    0 =< Toto < 1
```

En fait, on se rend compte avec un tout petit peu de pratique que cette fonction Aléa peut nous servir pour générer n'importe quel nombre compris dans n'importe quelle fourchette :

- si Alea génère un nombre compris entre 0 et 1, Alea multiplié par Z produit un nombre entre 0 et Z. Donc, il faut estimer la « largeur » de la fourchette voulue et multiplier Alea par cette « largeur » désirée.
- ensuite, si la fourchette ne commence pas à zéro, il va suffire d'ajouter ou de retrancher quelque chose pour « caler » la fourchette au bon endroit.

Par exemple, si je veux générer un nombre entre 1,35 et 1,65 ; la « fourchette » mesure 0,30 de large. Donc : 0 =< Alea()*0,30 < 0,30

Il suffit dès lors d'ajouter 1,35 pour obtenir la fourchette voulue. Si j'écris que :

```
Toto ← Alea()*0,30 + 1,35 ;
```

Toto aura bien une valeur comprise entre 1,35 et 1,65.

9.4. LES FONCTIONS DE CONVERSION

Dernière grande catégorie de fonctions, là aussi disponibles dans tous les langages, car leur rôle est parfois incontournable, les fonctions dites de conversion.

Tous les langages proposent une palette de fonctions destinées à opérer des conversions. On trouvera au moins une fonction destinée à convertir une chaîne en numérique (appelons-la Cnum en pseudo-code), et une convertissant un nombre en caractère (Ccar).

CHAPITRE 10
LES FICHIERS

Les **fichiers servent à stocker des informations de manière permanente, entre deux exécutions d'un programme**. Si les variables, qui sont des adresses de mémoire vive, disparaissent à chaque fin d'exécution, les fichiers, eux sont stockés sur des périphériques à mémoire de masse (disquette, disque dur, CD Rom…).

10.1. ORGANISATION DES FICHIERS

Deux catégories de fichiers :

- **le fichier organisé sous forme de lignes successives, il** contient le même genre d'information à chaque ligne. Ces lignes sont alors appelées des **enregistrements**. Un fichier ainsi codé sous forme d'enregistrements est appelé un **fichier texte**.
- le fichier qui ne possède pas de structure de lignes (d'enregistrement). Ces fichiers sont appelés des fichiers **binaires**.

10.2. STRUCTURE DES ENREGISTREMENTS

Deux possibilités :

- la **délimitation :** L'avantage de cette structure est son **faible encombrement en place mémoire** ; il n'y a aucun espace perdu, et un fichier texte codé de cette manière occupe le minimum de place possible. Mais elle possède en revanche un inconvénient majeur, qui est la **lenteur de la lecture**. En effet, chaque fois que l'on récupère une ligne dans le fichier, il faut alors parcourir un par un tous les caractères pour repérer chaque occurrence du caractère de séparation avant de pouvoir découper cette ligne en différents champs.
- **les champs de largeur fixe** : cette structure **gaspille de la place mémoire**, puisque le fichier est un vrai gruyère plein de trous. Mais d'un autre côté, la récupération des différents champs est très **rapide**. Lorsqu'on récupère une ligne, il suffit de la découper en différentes chaînes de longueur prédéfinie, et le tour est joué.

10.3. TYPES D'ACCES

On distingue :

- **L'accès séquentiel** : on ne peut accéder qu'à la donnée suivant celle qu'on vient de lire. On ne peut donc accéder à une information qu'en ayant au préalable examiné celle qui la précède. Dans le cas d'un fichier texte, cela signifie qu'on lit le fichier ligne par ligne (enregistrement par enregistrement).
- **L'accès direct** (ou **aléatoire**) : on peut accéder directement à l'enregistrement de son choix, en précisant le numéro de cet enregistrement. Mais cela veut souvent dire une gestion fastidieuse des déplacements dans le fichier.
- **L'accès indexé** : pour simplifier, il combine la rapidité de l'accès direct et la simplicité de l'accès séquentiel (en restant toutefois plus compliqué). Il est particulièrement adapté au traitement des gros fichiers, comme les bases de données importantes.

A la différence de la précédente, cette typologie **ne caractérise pas la structure** elle-même du fichier. En fait, tout fichier peut être utilisé avec l'un ou l'autre des trois types d'accès. Le choix du type d'accès n'est pas un choix qui concerne le fichier lui-même, mais uniquement la manière dont il va être traité par la machine. **C'est donc dans le programme, et seulement dans le programme, que l'on choisit le type d'accès souhaité.**

Pour conclure sur tout cela, voici un petit tableau récapitulatif :

	Fichiers Texte	Fichiers Binaires
On les utilise pour stocker...	des bases de données	tout, y compris des bases de données.
Ils sont structurés sous forme de...	lignes (enregistrements)	Ils n'ont pas de structure apparente. Ce sont des octets écrits à la suite les uns des autres.
Les données y sont écrites...	exclusivement en tant que caractères	comme en mémoire vive
Les enregistrements sont eux-mêmes structurés...	au choix, avec un séparateur ou en champs de largeur fixe	en champs de largeur fixe, s'il s'agit d'un fichier codant des enregistrements
Lisibilité	Le fichier est lisible clairement avec n'importe quel éditeur de texte	Le fichier a l'apparence d'une suite d'octets illisibles
Lecture du fichier	On ne peut lire le fichier que ligne par ligne	On peut lire les octets de son choix (y compris la totalité du fichier d'un coup)

Dans le cadre de ce cours, on se limitera volontairement au type de base : le fichier texte en accès séquentiel.

10.4. Instructions (fichiers texte en acces sequentiel)

Si l'on veut travailler sur un fichier, la première chose à faire est de l'**ouvrir**. Cela se fait en attribuant au fichier un **numéro de canal**. On ne peut ouvrir qu'un seul fichier par canal, mais quel que soit le langage, on dispose toujours de plusieurs canaux.

L'important est que lorsqu'on ouvre un fichier, on stipule ce qu'on va en faire : **lire, écrire ou ajouter.**

- Si on ouvre un fichier **pour lecture**, on pourra uniquement récupérer les informations qu'il contient, sans les modifier en aucune manière.
- Si on ouvre un fichier **pour écriture**, on pourra mettre dedans toutes les informations que l'on veut. Mais les informations précédentes, si elles existent, seront **intégralement écrasées** Et on ne pourra pas accéder aux informations qui existaient précédemment.
- Si on ouvre un fichier **pour ajout**, on ne peut ni lire, ni modifier les informations existantes. Mais on pourra ajouter de nouvelles lignes (**enregistrements)**.

Pour ouvrir un fichier texte, on écrira par exemple :

```
Ouvrir ("Exemple.txt", 4) en Lecture
```

Ici, "Exemple.txt" est le nom du fichier sur le disque dur, 4 est le numéro de canal, et ce fichier a donc été ouvert en lecture.

```
Car Truc, Nom, Prénom, Tel, Mail ;
Ouvrir ("Exemple.txt", 4) en Lecture ;
LireFichier 4, Truc
Nom ← Mid(Truc, 1, 20) ;
```

```
Prénom ← Mid(Truc, 21, 15) ;
Tel ← Mid(Truc, 36, 10) ;
Mail ← Mid(Truc, 46, 20) ;
```

L'instruction LireFichier récupère donc dans la variable spécifiée l'enregistrement suivant dans le fichier... "suivant", oui, mais par rapport à quoi ? Par rapport au dernier enregistrement lu. C'est en cela que le fichier est dit séquentiel. En l'occurrence, on récupère donc la première ligne, donc, le premier enregistrement du fichier, dans la variable Truc. Ensuite, le fichier étant organisé sous forme de champs de largeur fixe, il suffit de tronçonner cette variable Truc en autant de morceaux qu'il y a de champs dans l'enregistrement, et d'envoyer ces tronçons dans différentes variables.

Lire un fichier séquentiel de bout en bout suppose de programmer une **boucle**. Comme on sait rarement à l'avance combien d'enregistrements comporte le fichier, la solution consiste à utiliser la fonction EOF (acronyme pour End Of File). Cette fonction renvoie la valeur Vrai si on a atteint la fin du fichier (auquel cas une lecture supplémentaire déclencherait une erreur). L'algorithme, ultra classique, en pareil cas est donc :

```
Car Truc ;
Ouvrir ("Exemple.txt", 5) en Lecture
Tantque Non EOF(5)
  LireFichier 5, Truc ;
  …
FinTantQue
Fermer 5 ;
Fin
```

Si l'on veut stocker au fur et à mesure en mémoire vive les informations lues dans le fichier, on a recours à un ou plusieurs **tableaux**. Et comme on ne sait pas d'avance combien il y aurait d'enregistrements dans le fichier, on ne sait pas davantage combien il doit y avoir d'emplacements dans les tableaux. Les tableaux dynamiques interviennent dans ce cas.

En rassemblant l'ensemble des connaissances acquises, nous pouvons donc écrire le prototype du code qui effectue la lecture intégrale d'un fichier séquentiel, tout en recopiant l'ensemble des informations en mémoire vive :

```
Debut
Tableaux caractère Nom(), Prénom(), Tel(), Mail() ;
Ouvrir ("Exemple.txt", 5) en Lecture
i ← -1 ;
Tantque Non EOF(5)
  LireFichier 5, Truc ;
  i ← i + 1 ;
  Redim Nom(i) ;
  Redim Prénom(i) ;
  Redim Tel(i) ;
  Redim Mail(i) ;
  Nom(i) ← Mid(Truc, 1, 20) ;
  Prénom(i) ← Mid(Truc, 21, 15) ;
  Tel(i) ← Mid(Truc, 36, 10) ;
  Mail(i) ← Mid(Truc, 46, 20) ;
FinTantQue
Fermer 5 ;
Fin
```

Ici, on a fait le choix de recopier le fichier dans quatre tableaux distincts. On aurait pu également tout recopier dans un seul tableau : chaque case du tableau aurait alors été occupée par

une ligne complète (un enregistrement) du fichier. Cette solution nous aurait fait gagner du temps au départ, mais elle alourdit ensuite le code, puisque chaque fois que l'on a besoin d'une information au sein d'une case du tableau, il faudra aller procéder à une extraction via la fonction MID. Ce qu'on gagne par un bout, on le perd donc par l'autre.

Pour une opération d'écriture, ou d'ajout, il faut d'abord impérativement, constituer une chaîne équivalente à la nouvelle ligne du fichier. Cette chaîne doit donc être « calibrée » de la bonne manière, avec les différents champs qui « tombent » aux emplacements corrects:

```
Ouvrir ("Exemple.txt", 3) en Ajout ;
Caractère Truc ;
Caractère Nom*20, Prénom*15, Tel*10, Mail*20 ;
```

Une telle déclaration assure que quel que soit le contenu de la variable Nom, par exemple, celle-ci comptera toujours 20 caractères. Si son contenu est plus petit, alors un nombre correct d'espaces sera automatiquement ajouté pour combler. Si on tente d'y entrer un contenu trop long, celui-ci sera automatiquement tronqué. Voyons la suite :

```
Nom ← "ali" ;
Prénom ← "Madjid" ;
Tel ← "0348946532" ;
Mail ← ali@chaabi.dz ;
Truc ← Nom & Prénom & Tel & Mail ;
EcrireFichier 3, Truc ;
```

Et pour finir, une fois qu'on en a terminé avec un fichier, il ne faut pas oublier de fermer ce fichier. On libère ainsi le canal qu'il occupait (et accessoirement, on pourra utiliser ce canal dans la suite du programme pour un autre fichier… ou pour le même).

10.5. STRATEGIES DE TRAITEMENT

Il existe globalement deux manières de traiter les fichiers textes :

- l'une consiste à s'en tenir au fichier proprement dit, c'est-à-dire à modifier **directement** les informations sur le disque dur. C'est parfois un peu acrobatique, lorsqu'on veut supprimer un élément d'un fichier : on programme alors une boucle avec un test, qui recopie dans un deuxième fichier tous les éléments du premier fichier sauf un ; et il faut ensuite recopier intégralement le deuxième fichier à la place du premier fichier.
- l'autre stratégie consiste à **passer par un ou plusieurs tableaux**. En fait, le principe fondamental de cette approche est de commencer, avant toute autre chose, par **recopier l'intégralité du fichier de départ en mémoire vive**. Ensuite, on ne manipule que cette mémoire vive (concrètement, un ou plusieurs tableaux). Et lorsque le traitement est terminé, on recopie à nouveau dans l'autre sens, depuis la mémoire vive vers le fichier d'origine.

Les avantages de la seconde technique sont nombreux :

- la **rapidité** : les accès en mémoire vive sont des milliers de fois plus rapides (nanosecondes) que les accès aux mémoires de masse (millisecondes au mieux pour un

disque dur). En basculant le fichier du départ dans un tableau, on minimise le nombre ultérieur d'accès disque, tous les traitements étant ensuite effectués en mémoire.

- la **facilité de programmation** : bien qu'il faille écrire les instructions de recopie du fichier dans le tableau, pour peu qu'on doive tripoter les informations dans tous les sens, c'est largement plus facile de faire cela avec un tableau qu'avec des fichiers.

10.6. DONNEES STRUCTUREES

10.6.1 Données structurées simples

Jusqu'à présent, voilà comment se présentaient nos possibilités en matière de mémoire vive : nous pouvions réserver un emplacement pour une information d'un certain type. Un tel emplacement s'appelle une variable (quand vous en avez assez de me voir radoter, vous le dites). Nous pouvions aussi réserver une série d'emplacement numérotés pour une série d'informations **de même type**. Un tel emplacement s'appelle un tableau (même remarque).

Voici maintenant que nous pouvons réserver **une série d'emplacements pour des données de type différents**. Un tel emplacement s'appelle une **variable structurée**. Son utilité, lorsqu'on traite des fichiers texte (et même, des fichiers en général), saute aux yeux : car on va pouvoir calquer chacune des lignes du fichier en mémoire vive, et considérer que chaque enregistrement sera recopié dans une variable et une seule, qui lui sera adaptée. Ainsi, le problème du "découpage" de chaque enregistrement en différentes variables (le nom, le prénom, le numéro de téléphone, etc.) sera résolu d'avance, puisqu'on aura une structure, un gabarit, en quelque sorte, tout prêt d'avance pour accueillir et prédécouper nos enregistrements.

Toutefois, lorsque nous utilisions des variables de type prédéfini, comme des entiers, des booléens, etc. nous n'avions qu'une seule opération à effectuer : déclarer la variable en utilisant un des types existants. A présent que nous voulons créer un nouveau type de variable (par assemblage de types existants), il va falloir faire deux choses : d'abord, créer le type. Ensuite seulement, déclarer la (les) variable(s) d'après ce type.

Nous allons donc, avant même la déclaration des variables, créer un type, une structure, calquée sur celle de nos enregistrements, et donc prête à les accueillir :

```
Structure test
  Caractère * 20 Nom ;
  Caractère * 15 Prénom ;
  Caractère * 10 Tel ;
  Caractère * 20 Mail ;
Fin Structure
```

Ici, test est le nom de la structure. Ce mot jouera par la suite dans le programme exactement le même rôle que les types prédéfinis comme Entier, Caractère ou Booléen. Maintenant que la structure est définie, on va pouvoir, dans la section du programme où s'effectuent les déclarations, créer une ou des variables correspondant à cette structure :

```
Test Individu ;
```

On pourra remplir les différentes informations contenues au sein de la variable Individu de la manière suivante :

```
Individu ← "ali", "Madjid", "0348946532", ali@chaabi.dz;
```

On peut aussi avoir besoin d'accéder à un seul des champs de la variable structurée. Dans ce cas, on emploie le point :

```
Individu.Nom ← "ali" ;
Individu.Prénom ← "Madjid" ;
Individu.Tel ← "0348946532" ;
Individu.Mail ← ali@chaabi.dz;
```

Ainsi, écrire correctement une information dans le fichier est un simple, puisqu'on dispose d'une variable Individu au bon gabarit. Une fois remplis les différents champs de cette variable - ce qu'on vient de faire -, il n'y a plus qu'à envoyer celle-ci directement dans le fichier.

```
EcrireFichier (3, Individu) ;
```

De la même manière, dans l'autre sens, lorsque l'on effectue une opération de lecture dans le fichier Adresses, la chose sera considérablement simplifiée : la structure étant faite pour cela, on peut dorénavant se contenter de recopier une ligne du fichier dans une variable de type test. Pour charger l'individu suivant du fichier en mémoire vive, il suffira donc d'écrire :

```
LireFichier (5, Individu) ;
```

Et là, direct, nous avons quatre renseignements accessibles dans les quatre champs de la variable individu. Tout cela, évidemment, parce que la structure de notre variable Individu correspond parfaitement à la structure des enregistrements de notre fichier.

10.6.2 Tableaux de données structurées

Si à partir des types simples, on peut créer des variables et des tableaux de variables, à partir des types structurés, on peut créer des variables structurées... et des **tableaux de variables structurées**.

Cela veut dire que nous disposons d'une manière de gérer la mémoire vive qui va correspondre exactement à la structure d'un fichier texte (d'une base de données). Comme les structures se correspondent parfaitement, le nombre de manipulations à effectuer, autrement dit de lignes de programme à écrire, va être réduit au minimum. En fait, dans notre tableau structuré, les champs des emplacements du tableau correspondront aux champs du fichier texte, et les indices des emplacements du tableau correspondront aux différentes lignes du fichier.

Voici, à titre d'illustration, l'algorithme complet de lecture du fichier Adresses et de sa recopie intégrale en mémoire vive, en employant un tableau structuré.

```
Début
Structure test
Caractère * 20 Nom ;
Caractère * 15Prénom ;
Caractère * 10 Tel ;
Caractère * 20 Mail ;
Fin Structure
Tableau test Montableau() ;
Ouvrir ("Exemple.txt", 3) en Lecture ;
i ← -1 ;
Tantque Non EOF(3)
   i ← i + 1 ;
   Redim Montableau(i)
   LireFichier (3, Montableau(i)) ;
FinTantQue
Fermer 3 ;
Fin
```

Si nous voulons écrire, à un moment, le mail de l'individu n°13 du fichier à l'écran, il nous suffirait de passer l'ordre :

```
Ecrire Montableau(13).Mail ;
```

10.7. RECAPITULATIF GENERAL

Lorsqu'on est amené à travailler avec des données situées dans un fichier, plusieurs choix, en partie indépendants les uns des autres, doivent être faits :

- sur l'organisation en **enregistrements** du fichier (choix entre **fichier texte** ou **fichier binaire**)
- sur le mode d'**accès** aux enregistrements du fichier (**direct** ou **séquentiel**)
- sur l'organisation des **champs** au sein des enregistrements (présence de **séparateurs** ou **champs de largeur fixe**)
- sur la **méthode de traitement** des informations (recopie intégrale préalable du fichier en mémoire vive ou non)
- sur le **type de variables** utilisées pour cette recopie en mémoire vive (plusieurs tableaux de type **simple**, ou un seul tableau de type **structuré**).

Chacune de ces options présente avantages et inconvénients, et il est impossible de donner une règle de conduite valable en toute circonstance. Il faut connaître ces techniques, et savoir choisir la bonne option selon le problème à traiter.

CHAPITRE 11
PROCEDURES ET FONCTIONS

11.1. FONCTIONS PERSONNALISEES

11.1.1 Définitions

Une application peut procéder aux mêmes traitements à plusieurs endroits de son déroulement.

La manière la plus évidente de programmer ce genre de choses, c'est de répéter le code correspondant autant de fois que nécessaire. Ce n'est pal la meilleure solution.

Il faut opter pour une autre stratégie, qui consiste à séparer ce traitement du corps du programme et à regrouper les instructions qui le composent en un module séparé. Il ne restera alors plus qu'à appeler ce groupe d'instructions à chaque fois qu'on en a besoin. Ainsi, la lisibilité est assurée ; le programme devient **modulaire**, et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité de l'application.

Le corps du programme s'appelle alors le **programme principal**, et ces groupes d'instructions auxquels on a recours s'appellent des **fonctions** et des **procédures**.

11.1.2 Passage d'arguments

En langage algorithmique, on nomme les paramètres de la fonction les **arguments de la fonction**. Une fonction sera dorénavant déclarée comme suit :

```
Fonction Nom_fonction(Arguments formels) ;
```

Quant aux appels, ils s'écrivent :

```
Var ← Nom_fonction(arguments effectifs) ;
```

On peut passer autant d'arguments qu'on veut, et créer des fonctions avec deux, trois, ou quatre arguments ;

11.2. PROCEDURES

11.2.1 Généralités

Il peut arriver que dans un programme, on ait à réaliser des tâches répétitives, mais que ces tâches n'aient pas pour rôle de générer une valeur particulière, ou qu'elles aient pour rôle d'en générer plus d'une à la fois.

On pourrait se dire qu'il faut créer une fonction pour faire cela. Mais quelle serait la valeur renvoyée par la fonction ? Une fonction ne peut renvoyer qu'une seule valeur à la fois.

La solution consiste à utiliser des **procédures**.

En fait, **les fonctions ne sont finalement qu'un cas particulier des procédures, celui où doit être renvoyé vers la procédure appelante une valeur et une seule**. Dans tous les autres cas (celui où on ne renvoie aucune valeur, comme celui ou en en renvoie plusieurs), il faut donc avoir recours non à la forme particulière et simplifiée (la fonction), mais à la forme générale (la procédure).

Voici comment se présente une procédure :

```
Procédure Nom_procedure(paramètres_entrée_formels, paramètres_sortie_formels) ;
...
Fin
```

Dans le programme principal, l'appel à la procédure Nom_procedure devient :

```
Appeler Nom_procedure(paramètres_entrée_effectifs, paramètres_sortie_effectifs) ;
```

11.2.2 Le problème des arguments

De même qu'avec les fonctions, les valeurs qui circulent depuis la procédure appelante vers la procédure appelée se nomment des **arguments**, ou des **paramètres d'entrée** de la procédure. Leur rôle est de transmettre une information depuis le code donneur d'ordres jusqu'au code sous-traitant. Unique différence dans une procédure, on peut être amené à vouloir renvoyer des résultats vers le programme principal. Ces résultats que la procédure doit transmettre à la procédure appelante devront donc eux aussi être véhiculés par des paramètres. Mais cette fois, il s'agira de paramètres fonctionnant dans l'autre sens (du sous-traitant vers le donneur d'ordres), on les appellera donc des **paramètres de sortie**.

Donc, **toute procédure possédant un et un seul paramètre de sortie peut également être écrite sous forme d'une fonction**.

11.2.3 Le passage de paramètres Il en existe deux :

- le passage **par valeur**
- le passage **par référence**

Dans une procédure, **un paramètre passé par valeur ne peut être qu'un paramètre d'entrée**.

Quand on transmet un paramètre par valeur, on est sûr que même en cas de bug dans la procédure, la valeur de la variable transmise ne sera jamais modifiée par erreur dans le programme principal.

Dans le passage par référence, **la variable d'entrée ne contiendra pas la valeur , mais son adresse, c'est-à-dire sa référence**. Dès lors, **toute modification de cette variable sera immédiatement redirigée**. Ce type de variable porte un nom : on l'appelle **un pointeur**. Tous les paramètres passés par référence sont des pointeurs.

Passer un paramètre par référence présente deux avantages. On gagne en occupation de place mémoire, puisque le paramètre en question ne recopie pas les informations envoyées par la procédure appelante, mais qu'il se contente d'en noter l'adresse, et **cela permet d'utiliser ce paramètre tant en lecture (en entrée) qu'en écriture (en sortie)**, puisque toute modification de la valeur du paramètre aura pour effet de modifier la variable correspondante dans la procédure appelante.

Nous pouvons résumer tout cela par un petit tableau :

	passage par valeur	**passage par référence**
utilisation en entrée	Oui	oui
utilisation en sortie	Non	oui

11.3. VARIABLES PUBLIQUES ET PRIVEES

Il existe un troisième moyen d'échanger des informations entre différentes procédures et fonctions : c'est de ne pas avoir besoin de les échanger, en faisant en sorte que ces procédures et fonctions partagent littéralement les mêmes variables. Cela suppose d'avoir recours à des variables particulières, lisibles et utilisables par n'importe quelle procédure ou fonction de l'application.

Par défaut, une variable est déclarée au sein d'une procédure ou d'une fonction. Elle est donc créée avec cette procédure, et disparaît avec elle. Durant tout le temps de son existence, une telle variable n'est visible que par la procédure qui l'a vu naître. Voilà pourquoi ces variables par défaut sont dites **privées**, ou **locales**.

Mais à côté de cela, il est possible de créer des variables qui certes, seront déclarées dans une procédure, mais qui du moment où elles existeront, seront des variables communes à toutes les procédures et fonctions de l'application. Avec de telles variables, le problème de la transmission des valeurs d'une procédure (ou d'une fonction) à l'autre ne se pose même plus : la variable Toto, existant pour toute l'application, est accessible et modifiable depuis n'importe quelle ligne de code de cette application. Plus besoin donc de la transmettre ou de la renvoyer. Une telle variable est alors dite **publique**, ou **globale**.

La manière dont la déclaration d'une variable publique doit être faite est évidemment fonction de chaque langage de programmation. En algorithmique, on pourra utiliser le mot-clé **Publique** :

```
Entier Publique Toto ;
```

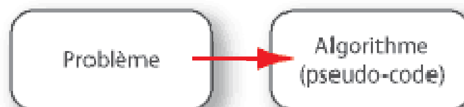
REMARQUES

Une application bien programmée est une application à l'architecture claire, dont les différents modules font ce qu'ils disent, disent ce qu'ils font, et peuvent être testés (ou modifiés) un par un sans perturber le reste de la construction. Il convient donc :

1. de **limiter au minimum l'utilisation des variables globales**.
2. de **regrouper sous forme de modules distincts** tous les morceaux de code qui possèdent une certaine unité fonctionnelle (programmation par "blocs").
3. de faire de ces modules **des fonctions lorsqu'ils renvoient un résultat unique**, et **des procédures dans tous les autres cas.**

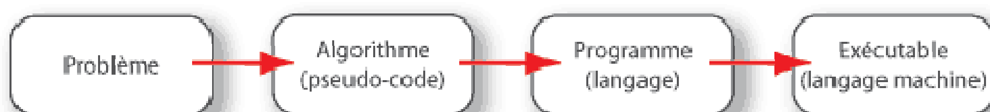
CHAPITRE 12
NOTIONS COMPLEMENTAIRES

12.1 INTERPRETATION ET COMPILATION



Si l'algorithme est bien écrit, sans faute logique, l'étape suivante ne doit normalement poser aucun problème conceptuel. Il n'y a plus qu'à effectuer une simple traduction.

A partir de là, le travail du programmeur est virtuellement terminé (en réalité, il reste tout de même une inévitable phase de tests, de corrections, etc., qui s'avère souvent très longue). Mais en tout cas, pour l'ordinateur, c'est là que les ennuis commencent. En effet, aucun ordinateur n'est en soi apte à exécuter les instructions telles qu'elles sont rédigées dans tel ou tel langage ; l'ordinateur, lui, ne comprend qu'un seul langage, qui est un langage codé en binaire et qui s'appelle le langage machine (ou assembleur).



C'est à cela que sert un langage : à vous épargner la programmation en binaire.

C'est pourquoi tout langage, à partir d'un programme écrit, doit obligatoirement procéder à une **traduction** en langage machine pour que ce programme soit exécutable.

Il existe deux stratégies de traduction, ces deux stratégies étant parfois disponibles au sein du même langage.

- le langage traduit les instructions au fur et à mesure qu'elles se présentent. Cela s'appelle la **compilation à la volée**, ou **l'interprétation**.
- le langage commence par traduire l'ensemble du programme en langage machine, constituant ainsi un deuxième programme (un deuxième fichier) distinct physiquement et logiquement du premier. Ensuite, et ensuite seulement, il exécute ce second programme. Cela s'appelle la **compilation**

Il va de soi qu'un langage interprété est plus maniable : on peut exécuter directement son code - et donc le tester - au fur et à mesure qu'on le tape, sans passer à chaque fois par l'étape supplémentaire de la compilation. Mais il va aussi de soi qu'un programme compilé s'exécute beaucoup plus rapidement qu'un programme interprété.

Toute application destinée à un usage professionnel est forcément une application compilée.

12.2. UNE LOGIQUE VICELARDE : LA PROGRAMMATION RECURSIVE

La programmation des fonctions personnalisées a donné lieu à l'essor d'une logique un peu particulière, adaptée en particulier au traitement de certains problèmes mathématiques (ou de jeux) : la programmation récursive. Pour expliquer de quoi il retourne, nous allons reprendre un exemple : le calcul d'une factorielle.

Rappelez-vous : la formule de calcul de la factorielle d'un nombre n s'écrit :

n ! = 1 x 2 x 3 x … x n

Nous pouvons programmer cela avec une boucle Pour. Mais une autre manière de voir les choses, serait de dire que quel que soit le nombre n :

n ! = n x (n-1) !

Si l'on doit programmer cela, on peut alors imaginer une fonction Fact, chargée de calculer la factorielle. Cette fonction effectue la multiplication du nombre passé en argument par la factorielle du nombre précédent. Et cette factorielle du nombre précédent va bien entendu être elle-même calculée par la fonction Fact.

Autrement dit, on va créer une fonction qui pour fournir son résultat, **va s'appeler elle-même un certain nombre de fois**. C'est cela, la récursivité.

On s'arrête quand on arrive au nombre 1, pour lequel la factorielle est par définition 1.

Cela produit l'écriture suivante, un peu déconcertante certes, mais parfois très pratique :

```
Fonction Fact (N) ;
Si N = 0 alors
  Fact ← 1 ;
Sinon
  Fact ← Fact(N-1) * N
Finsi
Fin
```

Le processus récursif remplace en quelque sorte la boucle, c'est-à-dire un processus itératif. On remarque qu'on traite le problème à l'envers : on part du nombre, et on remonte à rebours jusqu'à 1 pour pouvoir calculer la factorielle. Cet effet de rebours est caractéristique de la programmation récursive.

Pour conclure sur la récursivité, trois remarques fondamentales.

- la programmation récursive, pour traiter certains problèmes, est **très économique pour le programmeur** ; elle permet de faire les choses correctement, en très peu d'instructions.
- en revanche, elle est **très dispendieuse de ressources machine**. Car à l'exécution, la machine va être obligée de créer autant de variables temporaires que de « tours » de fonction en attente.
- **tout problème formulé en termes récursifs peut également être formulé en termes itératifs**.