Introduction à l'Algorithmique

L'algorithmique est un terme d'origine arabe, comme algèbre, amiral ou zénith.
Ce n'est bien sûr pas une excuse pour massacrer son orthographe, ou sa prononciation.
Ainsi, l'algo n'est pas « rythmique », à la différence du bon rock'n roll.
L'algo n'est pas non plus « l'agglo ».
Alors, ne confondez pas l'algorithmique avec l'agglo rythmique, qui consiste à poser des parpaings en cadence.

1. Qu'est-ce que l'algomachin ?

Un algorithme, c'est une suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné.
Si l'algorithme est juste, le résultat est le résultat voulu, ex : un touriste qui vous demande son chemin se retrouve là où il voulait aller, sinon … votre touriste trouvera une autre personne pour le renseigné (votre client un autre prestataire).

Complétons toutefois cette définition. Après tout, en effet, si l'algorithme, comme on vient de le dire, n'est qu'une suite d'instructions menant celui qui l'exécute à résoudre un problème, pourquoi ne pas donner comme instruction unique : « résous le problème », et laisser l'interlocuteur se débrouiller avec ça ? A ce tarif, n'importe qui serait champion d'algorithmique sans faire aucun effort.

Le malheur est que justement, si le touriste vous demande son chemin, c'est qu'il ne le connaît pas.

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

2. Faut-il être matheux pour être bon en algorithmique ?

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

1. 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, si l'on y tient, qu'intervient la forme « d'intelligence » requise pour l'algorithmique. Alors, c'est certain, il y a des gens qui possèdent au départ davantage cette intuition que les autres. Cependant 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 ».
2. 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. Cette opération ne requiert pas la moindre once d'intelligence. Mais elle reste néanmoins indispensable, si l'on ne veut pas écrire à l'aveuglette.

3. Algorithmique et programmation.

L'algorithmique exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage. Pour prendre une image, 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 Visual Basic, etc.) on doit en plus se colleter les problèmes de syntaxe, ou de types d'instructions, propres à ce langage.

A cela, il faut ajouter que des générations de programmeurs, souvent autodidactes (mais pas toujours, hélas !), ayant directement appris à programmer dans tel ou tel langage, ne font pas mentalement clairement la différence entre ce qui relève de la structure logique générale de toute programmation (les règles fondamentales de l'algorithmique) et ce qui relève du langage particulier qu'ils ont appris. Ces programmeurs, non seulement ont beaucoup plus de mal à passer ensuite à un langage différent, mais encore écrivent bien souvent des programmes qui même s'ils sont justes, restent laborieux. Car on n'ignore pas impunément les règles fondamentales de l'algorithmique… Alors, autant l'apprendre en tant que telle !

4. Avec quelles conventions écrit-on un algorithme ?

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.

Les Variables

1. A quoi servent les variables ?

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier), ou que sais-je encore. Il peut aussi s'agir de résultats obtenus par le programme, intermédiaires ou définitifs. Ces données peuvent être de plusieurs types (on en reparlera) : elles peuvent être des nombres, du texte, etc. Toujours est-il que dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une **variable**.

Pour employer une image, une variable est une boîte, que le programme (l'ordinateur) va repérer par une étiquette. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette.
En réalité, dans la mémoire vive de l'ordinateur, il n'y a bien sûr pas une vraie boîte, et pas davantage de vraie étiquette collée dessus (j'avais bien prévenu que la boîte et l'étiquette, c'était une image). Dans l'ordinateur, physiquement, il y a un emplacement de mémoire, repéré par une adresse binaire. Si on programmait dans un langage directement compréhensible par la machine, on devrait se fader de désigner nos données par de superbes 10011001 et autres 01001001 (enchanté !). Mauvaise nouvelle : de tels langages existent ! Ils portent le doux nom d'assembleur. Bonne nouvelle : ce ne sont pas les seuls langages disponibles.
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.

2. Déclaration des variables.

Le nom de la variable (l'étiquette de la boîte) **obéit à des impératifs changeant selon les langages**. Toutefois, 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. Un nom de variable correct commence également impérativement par une lettre. Quant au nombre maximal de signes pour un nom de variable, il dépend du langage utilisé.

En pseudo-code algorithmique, on est bien sûr libre du nombre de signes pour un nom de variable, même si pour des raisons purement pratiques on évite généralement les noms à rallonge.
Lorsqu'on déclare une variable, il ne suffit pas de créer une boîte (réserver un emplacement mémoire) ; encore doit-on préciser ce que l'on voudra mettre dedans, car de cela dépendent la taille de la boîte (**de l'emplacement mémoire**) et le type de codage utilisé.

1. Type numériques classiques :

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

En algorithmique, on ne se tracassera pas trop avec les sous-types de variables numériques (entier, double, position). On se contentera donc de préciser qu'il s'agit d'un nombre, en gardant en tête que dans un vrai langage, il faudra être plus précis.

En pseudo-code, une déclaration de variables aura ainsi cette tête :

```
Variable g en Numérique
```

Ou encore

```
Variables g, h, i, j en Numérique
```

2. 2 autres types numériques :

- Le type **date** (jour/mois/année)
- Le type **monétaire** (assez rare !)

3. Le type alphanumérique.

Fort heureusement, les boîtes que sont les variables peuvent contenir bien d'autres informations que des nombres. Sans cela, on serait un peu embêté dès que l'on devrait stocker un nom de famille, par exemple.

On dispose donc également du type **alphanumérique**, également appelé type **caractère**, type **chaîne** ou en anglais, le type **string**.

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 **string** dépend du langage utilisé.

Un groupe de caractères (y compris un groupe de un, ou de zéro caractères), qu'il soit ou non stocké dans une variable, d'ailleurs, est donc souvent appelé chaîne de caractères.

En pseudo-code, une chaîne de caractères est toujours notée entre guillemets

4. Le type booléen

Le dernier type de variables est le type booléen : on y stocke uniquement les valeurs logiques VRAI et FAUX.

On peut représenter ces notions abstraites de VRAI et de FAUX par tout ce qu'on veut : de l'anglais (TRUE et FALSE) ou des nombres (0 et 1). Peu importe. Ce qui compte, c'est de comprendre que le type booléen est très économique en termes de place mémoire occupée, puisque pour stocker une telle information binaire, un seul bit suffit.

3. L'instruction d'affectation

1. Syntaxe et signification

La seule chose qu'on puisse faire avec une variable, c'est **l'affecter**, c'est-à-dire **lui attribuer une valeur**. Pour poursuivre la superbe métaphore filée déjà employée, on peut « remplir la boîte ».

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

Ainsi :

```
Toto ← 24
```

Attribue la valeur 24 à la variable Toto.

Ceci, soit dit en passant, sous-entend impérativement que Toto soit une variable de type numérique. Si Toto a été défini dans un autre type, il faut bien comprendre que cette instruction provoquera une erreur.

On peut également sans aucun problème attribuer à une variable la valeur d'une autre variable, telle quelle ou modifiée. Par exemple :

```
Tutu ← Toto
```

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

Notez bien que cette instruction n'a en rien modifié la valeur 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. De même que précédemment, 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.

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

```
Variable A en Numérique
Début
A ← 34
A ← 12
Fin
```

Exemple 2

```
Variable A en Numérique
Début
A ← 12
A ← 34
Fin
```

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

4. Expressions et opérateurs

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

Cette définition vous paraît peut-être obscure. Mais réfléchissez-y quelques minutes, et vous verrez qu'elle recouvre quelque chose d'assez simple sur le fond. Par exemple, voyons quelques expressions de **type numérique**. Ainsi :

```
7
5+4
123-45+844
Toto-12+5-Riri
```

…sont toutes des expressions valides, pour peu que Toto et Riri soient bien des nombres. Car dans le cas contraire, la quatrième expression n'a pas de sens. En l'occurrence, les opérateurs que j'ai employés sont l'addition (+) et la soustraction (-).

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

a. Les opérateurs numériques.

Ce sont les quatre opérations arithmétiques tout ce qu'il y a de classique :

+ : 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. Pourquoi dès lors se fatiguer à mettre des parenthèses inutiles ?

En revanche, 12 * (3 + 5) vaut 12 * 8 soit 96. Rien de difficile là-dedans, que du normal.

b. Opérateur alphanumérique : &

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

```
Variables A, B, C en Caractère
Début
A ← "Gloubi"
B ← "Boulga"
C ← A & B
Fin
```

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

c. Opérateurs logiques (ou booléens) :

Il s'agit du ET, du OU, du NON et du mystérieux (mais rarissime XOR). Nous les laisserons de côté… provisoirement, soyez-en sûrs.

Nous les verrons plus en détails plus tard.

Lecture et Ecriture.

1. De quoi parle-t-on ?

Manipuler des variables en mémoire vive par un programme, c'est vrai que c'est très marrant, et d'ailleurs on a tous bien rigolé au chapitre précédent. Cela dit, à la fin de la foire, on peut tout de même se demander à quoi ça sert.

En effet. Imaginons que nous ayons fait un programme pour calculer le carré d'un nombre, mettons 12. Si on a fait au plus simple, on a écrit un truc du genre :

```
Variable A en Numérique
Début
A ← 12^2
Fin
```

D'une part, ce programme nous donne le carré de 12. C'est très gentil à lui. Mais si l'on veut le carré d'un autre nombre que 12, il faut réécrire le programme… bof
D'autre part, le résultat est indubitablement calculé par la machine. Mais elle le garde soigneusement pour elle, et le pauvre utilisateur qui fait exécuter ce programme, lui, ne saura jamais quel est le carré de 12. Re-bof.

C'est pourquoi, heureusement, il existe des d'instructions pour permettre à la machine de dialoguer avec l'utilisateur :

- Dans un sens, 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**.

2. Les instructions

Pour que l'utilisateur entre la (nouvelle) valeur de Titi, on mettra :

```
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 (sinon, le pauvre utilisateur passe son temps à se demander ce que l'ordinateur attend de lui… et c'est très désagréable !) :

```
Ecrire "Entrez votre nom : "
Lire NomFamille
```

Lecture et Ecriture sont des instructions algorithmiques qui ne présentent pas de difficultés particulières, une fois qu'on a bien assimilé ce problème du sens du dialogue (homme → machine, ou machine ← homme).

Les Tests conditionnels

1. De quoi s'agit-il ?

Reprenons le cas de notre « programmation algorithmique du touriste égaré ». Normalement, l'algorithme ressemblera à quelque chose comme : « *Allez tout droit jusqu'au prochain carrefour, puis prenez à droite et ensuite la deuxième à gauche, et vous y êtes* ».

Mais en cas de doute légitime de votre part, cela pourrait devenir : « *Allez tout droit jusqu'au prochain carrefour et là regardez à droite. Si la rue est autorisée à la circulation, alors prenez la et ensuite c'est la deuxième à gauche. Mais si en revanche elle est en sens interdit, alors continuez jusqu'à la prochaine à droite, prenez celle-là, et ensuite la première à droite* ».

Ce deuxième algorithme a ceci de supérieur au premier qu'il prévoit, en fonction d'une situation pouvant se présenter de deux façons différentes, deux façons différentes d'agir. Cela suppose que l'interlocuteur (le touriste) sache analyser la condition que nous avons fixée à son comportement (« la rue est-elle en sens interdit ? ») pour effectuer la série d'actions correspondante.

Heureusement les ordinateurs possèdent cette aptitude, sans laquelle d'ailleurs nous aurions bien du mal à les programmer. Nous allons donc pouvoir parler à notre ordinateur comme à notre touriste, et lui donner des séries d'instructions à effectuer selon que la situation se présente d'une manière ou d'une autre. Cette structure logique répond au doux nom de **test**.

2. Structure d'un test

Il n'y a que **deux formes possibles** pour un test ; la première est la plus simple, la seconde la plus complexe.

```
Si booléen Alors
    Instructions
Finsi
```

Et

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

Ceci appelle quelques explications :

Un **booléen** est une **expression** dont la valeur est VRAI ou FAUX. Cela peut donc être (il n'y a que deux possibilités) :

- une **variable** (ou une expression) de type booléen
- une **condition**

La machine exécutant le programme examine la valeur du booléen.

- Si ce booléen 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 le booléen est **FAUX**, l'ordinateur saute directement aux instructions situées après le Finsi.
  Dans le cas où le test contient un **Sinon**, et que le booléen est **FAUX**, alors l'ordinateur exécute la série d'instructions entre le Sinon et le Finsi.

Exprimé sous forme de pseudo-code, la programmation de notre touriste de tout à l'heure donnerait donc quelque chose du genre :

```
Allez tout droit jusqu'au prochain carrefour
Si la rue à droite est autorisée à la circulation Alors
  Tournez à droite
  Avancez
  Prenez la deuxième à gauche
Sinon
  Continuez jusqu'à la prochaine rue à droite
  Prenez cette rue
  Prenez la première à droite
Finsi
```

3. Qu'est-ce qu'une condition ?

Une condition est une comparaison.

Cette définition est essentielle ! 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 :

- égal à…
- différent de…
- strictement plus petit que…
- strictement plus grand que…
- plus petit ou égal à…
- plus grand ou égal à…

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**.

4. Conditions composées

Prenons l'exemple « Toto est-il 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**.

Comme on l'a évoqué plus haut, l'informatique met 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.

- Il faut se méfier un peu plus du OU. 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 OU informatique ne veut donc pas dire « ou bien »

- 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 XOR est donc l'équivalent du "ou bien" du langage courant. J'insiste toutefois sur le fait que le XOR est une rareté, dont il n'est pas strictement indispensable de s'encombrer en programmation.

- Enfin, le NON inverse une condition : NON(Condition1)est VRAI si Condition1 est FAUX, et il sera FAUX si Condition1 est VRAI. C'est l'équivalent pour les booléens du signe "moins" que l'on place devant les nombres.
  Alors, vous vous demandez peut-être à quoi sert ce NON. Après tout, plutôt qu'écrire NON(Prix > 20), il serait plus simple d'écrire tout bonnement Prix=<20. Dans ce cas précis, c'est évident qu'on se complique inutilement la vie avec le NON. Mais si le NON n'est jamais indispensable, il y a tout de même des situations dans lesquelles il s'avère bien utile.

On représente fréquemment tout ceci dans des **tables de vérité** (C1 et C2 représentent deux conditions, et on envisage à chaque fois les quatre cas possibles)

C1 et C2	C2 Vrai	C2 Faux
C1 Vrai	Vrai	Faux
C1 Faux	Faux	Faux

C1 ou C2	C2 Vrai	C2 Faux
C1 Vrai	Vrai	Vrai
C1 Faux	Vrai	Faux

C1 xor C2	C2 Vrai	C2 Faux
C1 Vrai	Faux	Vrai
C1 Faux	Vrai	Faux

Non C1	
C1 Vrai	Faux
C1 Faux	Vrai

5. Tests imbriqués

Imaginons 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 :

```
Variable Temp en Entier
Début
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
```

Vous constaterez que c'est un peu laborieux. Les conditions se ressemblent plus ou moins, et surtout 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 :

```
Variable Temp en Entier
Début
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
```

Nous avons fait des économies : au lieu de devoir taper trois conditions, dont une composée, nous n'avons plus que deux conditions simples. Mais aussi, et surtout, nous avons fait des économies sur le temps d'exécution de l'ordinateur. Si la température est inférieure à zéro, celui-ci écrit dorénavant « C'est de la glace » et passe **directement** à la fin, sans être ralenti par l'examen d'autres possibilités (qui sont forcément fausses).

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

Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

Dans le cas de tests imbriqués, le Sinon et le Si peuvent être fusionnés en un SinonSi. On considère alors qu'il s'agit d'un seul bloc de test, conclu par un seul FinSi

6. Variables Booléennes

Jusqu'ici, pour écrire nos des tests, nous avons utilisé uniquement des **conditions**. Mais vous vous rappelez qu'il existe un type de variables (les booléennes) susceptibles de stocker les valeurs VRAI ou FAUX. En fait, on peut donc entrer des conditions dans ces variables, et tester ensuite la valeur de ces variables.

Reprenons l'exemple de l'eau. On pourrait le réécrire ainsi :

```
Variable Temp en Entier
Variables A, B en Booléen
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
A ← Temp =< 0
B ← Temp < 100
Si A Alors
  Ecrire "C'est de la glace"
SinonSi B Alors
  Ecrire "C'est du liquide"
Sinon
  Ecrire "C'est de la vapeur"
Finsi
Fin
```

A priori, cette technique ne présente guère d'intérêt : on a alourdi plutôt qu'allégé l'algorithme de départ, en ayant recours à deux variables supplémentaires.

- Mais souvenons-nous : une variable booléenne n'a besoin que d'un seul bit pour être stockée. De ce point de vue, l'alourdissement n'est donc pas considérable.
- Dans certains cas, notamment celui de conditions composées très lourdes (avec plein de ET et de OU tout partout) cette technique peut faciliter le travail du programmeur, en améliorant nettement la lisibilité de l'algorithme. Les variables booléennes peuvent également s'avérer très utiles pour servir de **flag**, technique dont on reparlera plus loin (rassurez-vous, rien à voir avec le flagrant délit des policiers).

Les boucles.

1. A quoi cela sert-il donc ?

Prenons le cas d'une saisie au clavier (une lecture), où par exemple, le programme pose une question à laquelle l'utilisateur doit répondre par O (Oui) ou N (Non). Mais tôt ou tard, l'utilisateur, facétieux ou maladroit, risque de taper autre chose que la réponse attendue. Dès lors, le programme peut planter soit par une erreur d'exécution (parce que le type de réponse ne correspond pas au type de la variable attendu) soit par une erreur fonctionnelle (il se déroule normalement jusqu'au bout, mais en produisant des résultats fantaisistes).

Alors, dans tout programme un tant soit peu sérieux, on met en place ce qu'on appelle un **contrôle de saisie**, afin de vérifier que les données entrées au clavier correspondent bien à celles attendues par l'algorithme.

On pourrait essayer avec un SI. Voyons voir ce que ça donne :

```
Variable Rep en Caractère
Début
Ecrire "Voulez vous un café ? (O/N)"
Lire Rep
Si Rep <> "O" et Rep <> "N" Alors
  Ecrire "Saisie erronnée. Recommencez"
  Lire Rep
FinSi
Fin
```

C'est impeccable. Du moins tant que l'utilisateur a le bon goût de ne se tromper qu'une seule fois, et d'entrer une valeur correcte à la deuxième demande. Si l'on veut également bétonner en cas de deuxième erreur, il faudrait rajouter un SI. Et ainsi de suite… cela va devenir compliqué, car quand s'arrêter ?

La solution consistant à aligner des SI… en pagaille est donc une impasse. La seule issue est donc **une structure de boucle**, qui se présente ainsi :

```
TantQue booléen
    ...
    Instructions
    ...
FinTantQue
```

Le principe est simple : le programme arrive sur la ligne du **TantQue**. Il examine alors la valeur du booléen. 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…

Illustration avec notre problème de contrôle de saisie. Une première approximation de la solution consiste à écrire :

```
Variable Rep en Caractère
Début
Ecrire "Voulez vous un café ? (O/N)"
TantQue Rep <> "O" et Rep <> "N"
  Ecrire "Vous devez répondre par O ou N. Recommencez"
  Lire Rep
```

```
FinTantQue
Ecrire "Saisie acceptée"
Fin
```

2. Boucler en comptant, ou compter en bouclant.

Une boucle pouvait être utilisée pour augmenter la valeur d'une variable. Cette utilisation des boucles est très fréquente, et dans ce cas, 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 (puisque le nombre de tours dépend de la valeur d'un booléen).

C'est pourquoi une autre structure de boucle est à notre disposition : **Pour … Suivant :**

```
Variable Truc en Entier
Début
Truc ← 0
TantQue Truc < 15
  Truc ← Truc + 1
  Ecrire "Passage numéro : ", Truc
FinTantQue
Fin
```

Equivaut à :

```
Variable Truc en Entier
Début
Pour Truc ← 1 à 15
  Ecrire "Passage numéro : ", Truc
  Truc Suivant
Fin
```

Insistons : **la structure « Pour … Suivant » 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**).

Il faut noter que dans une structure « Pour … Suivant », 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 et en terminant par la deuxième.
Mais vous pouvez si vous le souhaitez préciser le **Pas** à utiliser pour votre incrémentation :

```
Pour Compteur ← Initial à Final Pas ValeurDuPas
...
Instructions
...
Compteur suivant
```

3. Les boucles imbriquées.

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. Il n'y a pas de raison :

```
Variables Truc, Trac en Entier
Début
Pour Truc ← 1 à 15
  Ecrire "Il est passé par ici"
  Pour Trac ← 1 à 6
    Ecrire "Il repassera par là"
  Trac Suivant
Truc Suivant
Fin
```

Dans cet exemple, le programme écrira une fois "il est passé par ici" puis six fois de suite "il repassera par là", et ceci quinze fois en tout. A la fin, il y aura donc eu 15 x 6 = 90 passages dans la deuxième boucle (celle du milieu), donc 90 écritures à l'écran du message « il repassera par là ».

Notez la différence marquante avec cette structure :

```
Variables Truc, Trac en Entier
Début
Pour Truc ← 1 à 15
  Ecrire "Il est passé par ici"
Truc Suivant
Pour Trac ← 1 à 6
  Ecrire "Il repassera par là"
Trac Suivant
Fin
```

Ici, il y aura quinze écritures consécutives de "il est passé par ici", puis six écritures consécutives de "il repassera par là", et ce sera tout.

Des boucles peuvent donc être imbriquées (cas n°1) ou successives (cas n°2).

Cependant, elles ne peuvent jamais, au grand jamais, être croisées.

4. Rappel important :

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, comme par exemple :

- le contrôle d'une saisie
- la gestion des tours d'un jeu (tant que la partie n'est pas finie, on recommence)
- la lecture des enregistrements d'un fichier de taille inconnue

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.

Les Tableaux

1. Utilité des tableaux

En programmation, un tableau permet de rassembler un ensemble de variables au sein d'une seule.

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 atrocité du genre :

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

C'est tout de même bigrement laborieux.
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. En bon français, cela donnerait donc quelque chose du genre « la note numéro 1 », « la note numéro 2 », « la note numéro 8 ». C'est largement plus pratique, vous vous en doutez.

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(ou pointer).
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.

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(0), Note(1), etc. **Attention, les indices des tableaux commencent généralement à 0, et non à 1**.

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).

En nous calquant sur les choix les plus fréquents dans les langages de programmations, nous déciderons ici arbitrairement et une bonne fois pour toutes que :

- les "cases" sont numérotées à partir de zéro, autrement dit que le plus petit indice est zéro.
- lors de la déclaration d'un tableau, on précise la plus grande valeur de l'indice (différente, donc, du nombre de cases du tableau, puisque si on veut 12 emplacements, le plus grand indice sera 11).

```
Tableau Note(11) en Entier
```

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, hormis dans quelques rares langages, 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 Note(11) en Numérique
Variables Moy, Som en Numérique
Début
Pour i ← 0 à 11
  Ecrire "Entrez la note n°", i
  Lire Note(i)
i Suivant
Som ← 0
Pour i ← 0 à 11
  Som ← Som + Note(i)
i Suivant
Moy ← Som / 12
Fin
```

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 0
- être un nombre entier
- être inférieure au nombre d'éléments du tableau

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, pourquoi pas, au diable la varice) 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**.

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.

Exemple : on veut faire saisir des notes pour un calcul de moyenne, mais on ne sait pas combien il y aura de notes à saisir. Le début de l'algorithme sera quelque chose du genre :

```
Tableau Notes() en Numérique
Variable nb en Numérique
Début
Ecrire "Combien y a-t-il de notes à saisir ?"
Lire nb
Redim Notes(nb-1)
```

4. Tableaux à 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, pour stocker par exemple des coordonnées se déclare ainsi :

```
Tableau Cases(7, 7) en Numérique
```

Cela veut dire : allouer 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.

5. Tableaux à n dimensions

Le principe est le même que pour 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.

Les Fonctions Prédéfinies

1. Généralités

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. C'est par exemple le cas du calcul du sinus d'un angle.
Si l'on souhaite stocker le sinus de 35 dans une variable, on procèdera ainsi :

```
A ← Sin(35)
```

En général, une fonction est donc constituée de trois parties :

- le **nom** proprement dit de la fonction. Ce nom ne s'invente pas ! Il doit impérativement correspondre à une fonction proposée par le langage de destination de votre algorithme.
- **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**.
  Notez également que les arguments doivent être d'un certain type, et qu'il faut respecter ces **types**.

2. Fonctions de texte classiques :

- **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.
- **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
- **Find(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 vices-et-versa (fonction **Chr**) :

- **Asc(chaîne)** : renvoi le code Ascii correspondant au caractère.
- **Chr(code)** : renvoi le caractère de la table Ascii correspondant au code (un entier).

3. Fonctions numériques classiques

- **Ent(double) :** reverra la partie entière d'un chiffre à décimales (Ent(3,228) renvoi 3)
- **Mod(entier,entier) :** renvoi le reste d'une division
  - Mod(10,3) vaut 1 car 10 = 3*3+1

    - Mod(12,2) vaut 0 car 12 = 6*2
- **Alea()** : renvoi un nombre aléatoire entre 0 et 1.

4. Fonctions personnalisées

a. De quoi s'agit-il ?

Une application, surtout si elle est longue, a toutes les chances de devoir procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits de son déroulement. Par exemple, la saisie d'une réponse par oui ou par non (et le contrôle qu'elle implique), peuvent être répétés dix fois à des moments différents de la même application, pour dix questions différentes.

Il faut alors opter pour 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 (qui n'existe donc désormais qu'en un exemplaire unique) 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 la **procédure principale**, et ces groupes d'instructions auxquels on a recours s'appellent des **fonctions** et des **sous-procédures** (nous verrons un peu plus loin la différence entre ces deux termes).

Une fonction est constituée d'un nom, du type de valeur qu'elle renvoi, et bien sûr d'une suite d'instructions, par exemple :

```
Fonction RepOuiNon() en caractère
Truc ← ""
TantQue Truc <> "Oui" et Truc <> "Non"
  Ecrire "Tapez Oui ou Non"
  Lire Truc
FinTantQue
Renvoyer Truc
Fin
```

On remarque au passage l'apparition d'un nouveau mot-clé : **Renvoyer**, qui indique quelle valeur doit prendre la fonction lorsqu'elle est utilisée par le programme.

Une fonction s'écrit toujours **en-dehors de la procédure principale**. Selon les langages, cela peut prendre différentes formes. Mais ce qu'il faut comprendre, c'est que ces quelques lignes de codes sont en quelque sorte des satellites, qui existent en dehors du traitement lui-même. Simplement, elles sont à sa disposition, et il pourra y faire appel chaque fois que nécessaire. Si l'on reprend notre exemple, une fois notre fonction RepOuiNon écrite, le programme principal comprendra les lignes :

```
Ecrire "Etes-vous marié ?"
Rep1 ← RepOuiNon()
Ecrire "Avez-vous des enfants ?"
Rep2 ← RepOuiNon()
```

b. Passage d'arguments

Reprenons l'exemple qui précède et analysons-le. On écrit un message à l'écran, puis on appelle la fonction RepOuiNon pour poser une question ; puis, un peu plus loin, on écrit un autre message à l'écran, et on appelle de nouveau la fonction pour poser la même question, etc. C'est une démarche acceptable, mais qui peut encore être améliorée : puisque avant chaque question, on doit écrire un message, autant que cette écriture du message figure directement dans la fonction appelée. Cela implique deux choses :

- lorsqu'on appelle la fonction, on doit lui préciser quel message elle doit afficher avant de lire la réponse.
- la fonction doit être « prévenue » qu'elle recevra un message, et être capable de le récupérer pour l'afficher.

En programmation, on dira que le « message » devient un **argument (ou un paramètre)** de la fonction.

La fonction sera alors dorénavant déclarée comme suit :

```
Fonction RepOuiNon(Msg en Caractère) en Caractère
Ecrire Msg
Truc ← ""
TantQue Truc <> "Oui" et Truc <> "Non"
  Ecrire "Tapez Oui ou Non"
  Lire Truc
FinTantQue
Renvoyer Truc
Fin Fonction
```

Il y a donc maintenant entre les parenthèses une variable, Msg, dont on précise le type, et qui signale à la fonction qu'un argument doit lui être envoyé à chaque appel. Quant à ces appels, justement, ils se simplifieront encore dans la procédure principale, pour devenir :

```
Rep1 ← RepOuiNon("Etes-vous marié ?")
Rep2 ← RepOuiNon("Avez-vous des enfants ?")
```

5. Sous-Procédures

Les fonctions, c'est bien, mais dans certains cas, ça ne nous rend guère service.

Il peut en effet 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.

En fait, les fonctions - que nous avons vues - ne sont finalement qu'un cas particulier des sous-procédures - que nous allons voir : 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 sous-procédure).

Parlons donc de ce qui est commun aux sous-procédures et aux fonctions, mais aussi de ce qui les différencie. Voici comment se présente une sous-procédure :

```
Procédure Bidule( ... )
  Instructions...
Fin Procédure
```

Dans la procédure principale, l'appel à la sous-procédure Bidule devient quant à lui :

```
Appeler Bidule(...)
```

L'on peut alors passer un argument à la procédure afin qu'il l'utilise, on dit alors que l'on passe l'argument **par valeur**.
Ou alors on passe un argument à la procédure afin que celle-ci en modifie la valeur afin que l'on puisse l'utiliser dans la procédure principale.

```
Procédure RepOuiNon(Msg en Caractère par valeur, Truc en Caractère par référence)
Ecrire Msg
Truc ← ""
TantQue Truc <> "Oui" et Truc <> "Non"
  Ecrire "Tapez Oui ou Non"
  Lire Truc
FinTantQue
Fin Fonction
```

Ce qui donne à l'utilisation :

```
M ← "Etes-vous marié ?"
Appeler RepOuiNon(M, T)
Ecrire "Votre réponse est ", T
```

Les Fichiers

Jusqu'à présent, les informations utilisées dans nos programmes ne pouvaient provenir que de deux sources : soit elles étaient inclues dans l'algorithme lui-même, par le programmeur, soit elles étaient entrées en cours de route par l'utilisateur. Mais évidemment, cela ne suffit pas à combler les besoins réels des informaticiens.

Imaginons que l'on veuille écrire un programme gérant un carnet d'adresses. D'une exécution du programme à l'autre, l'utilisateur doit pouvoir retrouver son carnet à jour, avec les modifications qu'il y a apportées la dernière fois qu'il a exécuté le programme. Les données du carnet d'adresse ne peuvent donc être inclues dans l'algorithme, et encore moins être entrées au clavier à chaque nouvelle exécution !

Les **fichiers** sont là pour combler ce manque. **Ils servent à stocker des informations de manière permanente, entre deux exécutions d'un programme**. Car si les variables, qui sont je le rappelle 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…).

1. Organisation des fichiers

Un premier grand critère, qui différencie les deux grandes catégories de fichiers, est le suivant : **le fichier est-il ou non organisé sous forme de lignes successives ?** Si oui, cela signifie vraisemblablement que ce fichier contient le même genre d'information à chaque ligne. Ces lignes sont alors appelées des **enregistrements**.

Afin d'illuminer ces propos obscurs, prenons le cas classique, celui d'un carnet d'adresses. Le fichier est destiné à mémoriser les coordonnées (ce sont toujours les plus mal chaussées, bien sûr) d'un certain nombre de personnes. Pour chacune, il faudra noter le nom, le prénom, le numéro de téléphone et l'email. Dans ce cas, il peut paraître plus simple de stocker une personne par ligne du fichier (par enregistrement). Dit autrement, quand on prendra une ligne, on sera sûr qu'elle contient les informations concernant une personne, et uniquement cela. Un fichier ainsi codé sous forme d'enregistrements est appelé un **fichier texte**.

En fait, entre chaque enregistrement, sont stockés les octets correspondants aux caractères CR (code Ascii 13) et LF (code Ascii 10), signifiant un retour au début de la ligne suivante. Le plus souvent, le langage de programmation, dès lors qu'il s'agit d'un fichier texte, gèrera lui-même la lecture et l'écriture de ces deux caractères à chaque fin de ligne : c'est autant de moins dont le programmeur aura à s'occuper. Le programmeur, lui, n'aura qu'à dire à la machine de lire une ligne, ou d'en écrire une.

Ce type de fichier est couramment utilisé dès lors que l'on doit stocker des informations pouvant être assimilées à une base de données.

Le second type de fichier, vous l'aurez deviné, se définit a contrario : il rassemble les fichiers qui ne possèdent pas de structure de lignes (d'enregistrement). Les octets, quels qu'ils soient, sont écrits à la queue leu leu. Ces fichiers sont appelés des fichiers **binaires**. Naturellement, leur structure différente implique un traitement différent par le programmeur. Tous les fichiers qui ne codent pas

une base de données sont obligatoirement des fichiers binaires : cela concerne par exemple un fichier son, une image, un programme exécutable, etc. Toutefois, on en dira quelques mots un peu plus loin, il est toujours possible d'opter pour une structure binaire même dans le cas où le fichier représente une base de données.

Autre différence majeure entre fichiers texte et fichiers binaires : dans un fichier texte, toutes les données sont écrites sous forme de... texte (étonnant, non ?). Cela veut dire que les nombres y sont représentés sous forme de suite de chiffres (des chaînes de caractères). **Ces nombres doivent donc être convertis en chaînes** lors de l'écriture dans le fichier. Inversement, lors de la lecture du fichier, on devra **convertir ces chaînes en nombre** si l'on veut pouvoir les utiliser dans des calculs. En revanche, dans les fichiers binaires, les données sont écrites à l'image exacte de leur codage en mémoire vive, ce qui épargne toutes ces opérations de conversion.

Ceci a comme autre implication qu'**un fichier texte est directement lisible**, alors qu'**un fichier binaire ne l'est pas** (sauf bien sûr en écrivant soi-même un programme approprié). Si l'on ouvre un fichier texte via un éditeur de textes, comme le bloc-notes de Windows, on y reconnaîtra toutes les informations (ce sont des caractères, stockés comme tels). La même chose avec un fichier binaire ne nous produit à l'écran qu'un galimatias de scribouillis incompréhensibles.

2. Structure des enregistrements

Savoir que les fichiers peuvent être structurés en enregistrements, c'est bien. Mais savoir comment sont à leur tour structurés ces enregistrements, c'est mieux. Or, là aussi, il y a deux grandes possibilités. Ces deux grandes variantes pour structurer les données au sein d'un fichier texte sont la **délimitation** et les **champs de largeur fixe**.

Reprenons le cas du carnet d'adresses, avec dedans le nom, le prénom, le téléphone et l'email. Les données, sur le fichier texte, peuvent être organisées ainsi :

```
Structure n°1
"Fonfec";"Sophie";0142156487;"fonfec@yahoo.fr"
"Zétofrais";"Mélanie";0456912347;"zétofrais@free.fr"
"Herbien";"Jean-Philippe";0289765194;"vantard@free.fr"
"Hergébel";"Octave";0149875231;"rg@aol.fr"
```

Ou ainsi :

```
Structure n°2
Fonfec          Sophie          0142156487fonfec@yahoo.fr
Zétofrais       Mélanie         0456912347zétofrais@free.fr
Herbien         Jean-Philippe   0289765194vantard@free.fr
Hergébel        Octave          0149875231rg@aol.fr
```

La structure n°1 est dite délimitée ; Elle utilise un caractère spécial, appelé caractère de délimitation, qui permet de repérer quand finit un champ et quand commence le suivant. Il va de soi que ce caractère de délimitation doit être strictement interdit à l'intérieur de chaque champ, faute de quoi la structure devient proprement illisible.

La structure n°2, elle, est dite à champs de largeur fixe. Il n'y a pas de caractère de délimitation, mais on sait que les x premiers caractères de chaque ligne stockent le nom, les y suivants le prénom, etc. Cela impose bien entendu de ne pas saisir un renseignement plus long que le champ prévu pour l'accueillir.

L'avantage de la structure n°1 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.

La structure n°2, à l'inverse, 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.

A l'époque où la place mémoire coûtait cher, la structure délimitée était souvent privilégiée. Mais depuis bien des années, la quasi-totalité des logiciels – et des programmeurs – optent pour la structure en champs de largeur fixe. Aussi, sauf mention contraire, nous ne travaillerons qu'avec des fichiers bâtis sur cette structure.

Remarque importante : lorsqu'on choisit de coder une base de données sous forme de champs de largeur fixe, on peut alors très bien opter pour un fichier binaire. Les enregistrements y seront certes à la queue leu leu, sans que rien ne nous signale la jointure entre chaque enregistrement. Mais si on sait combien d'octets mesure invariablement chaque champ, on sait du coup combien d'octets mesure chaque enregistrement. Et on peut donc très facilement récupérer les informations : si je sais que dans mon carnet d'adresse, chaque individu occupe mettons 75 octets, alors dans mon fichier binaire, je déduis que l'individu n°1 occupe les octets 1 à 75, l'individu n°2 les octets 76 à 150, l'individu n°3 les octets 151 à 225, etc.

3. Types d'accès

On vient de voir que l'organisation des données au sein des enregistrements du fichier pouvait s'effecteur selon deux grands choix stratégiques. Mais il existe une autre ligne de partage des fichiers : le type d'accès, autrement dit la manière dont la machine va pouvoir aller rechercher les informations contenues dans le fichier.

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	On peut lire les octets de son choix (y compris la totalité du

	ligne par ligne	fichier d'un coup)

4. Instructions (fichiers texte en accès séquentiel)

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, donc pas de soucis.

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, comme vous commencez à vous en douter, ajouter de nouvelles lignes (je rappelle qu'au terme de lignes, on préférera celui d'enregistrements.

Au premier abord, ces limitations peuvent sembler infernales. Au deuxième abord, elles le sont effectivement. Il n'y a même pas d'instructions qui permettent de supprimer un enregistrement d'un fichier !

Toutefois, avec un peu d'habitude, on se rend compte que malgré tout, même si ce n'est pas toujours marrant, on peut quand même faire tout ce qu'on veut avec ces fichiers séquentiels.

Pour ouvrir un fichier texte, on écrira par exemple :

```
Ouvrir "Exemple.txt" sur 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. Vous l'aviez sans doute pressenti. Allons plus loin :

```
Variables Truc, Nom, Prénom, Tel, Mail en Caractères
Début
Ouvrir "Exemple.txt" sur 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. Et le tour est joué.

La suite du raisonnement s'impose avec une logique impitoyable : 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 combine consiste neuf fois sur dix 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 :

```
Variable Truc en Caractère
Début
Ouvrir "Exemple.txt" sur 5 en Lecture
Tantque Non EOF(5)
  LireFichier 5, Truc
  …
FinTantQue
Fermer 5
Fin
```

Et neuf fois sur dix également, 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. Qu'importe, les programmeurs avertis que vous êtes connaissent la combine des tableaux dynamiques.

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 :

```
Tableaux Nom(), Prénom(), Tel(), Mail() en Caractère
Début
Ouvrir "Exemple.txt" sur 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.

Mais surtout, comme on va le voir bientôt, il y a autre possibilité, bien meilleure, qui cumule les avantages sans avoir aucun des inconvénients.

Néanmoins, ne nous impatientons pas, chaque chose en son temps, et revenons pour le moment à la solution que nous avons employée ci-dessus.

Pour une opération d'écriture, ou d'ajout, il faut d'abord impérativement, sous peine de semer la panique dans la structure du fichier, 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. Le moyen le plus simple pour s'épargner de longs traitements est de procéder avec des chaînes correctement dimensionnées dès leur déclaration (la plupart des langages offrent cette possibilité) :

```
Ouvrir "Exemple.txt" sur 3 en Ajout
Variable Truc en Caractère
Variables Nom*20, Prénom*15, Tel*10, Mail*20 en Caractère
```

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 • "Jokers"
Prénom • "Midnight"
Tel • "0348946532"
Mail • "allstars@rockandroll.com"
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).

5. Stratégies 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 (ou presque) 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, comme on l’a vu, 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, et 99 fois sur 100, c'est ainsi qu'il faudra procéder :

- 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.

Pourquoi, alors, demanderez-vous haletants, ne fait-on pas cela à tous les coups ? Y a-t-il des cas où il vaut mieux en rester aux fichiers et ne pas passer par des tableaux ?

La recopie d’un très gros fichier en mémoire vive exige des ressources qui peuvent atteindre des dimensions considérables. Donc, dans le cas d'immenses fichiers (très rares, cependant), cette recopie en mémoire peut s'avérer problématique.

Toutefois, lorsque le fichier contient des données de type non homogènes (chaînes, numériques, etc.) cela risque d’être coton pour le stocker dans un tableau unique : il va falloir déclarer plusieurs tableaux, dont le maniement au final peut être aussi lourd que celui des fichiers de départ.

A moins... d'utiliser une ruse : créer des types de variables personnalisés, composés d’un « collage » de plusieurs types existants (10 caractères, puis un numérique, puis 15 caractères, etc.). Ce type de variable s'appelle un type structuré. Cette technique, bien qu’elle ne soit pas vraiment difficile, exige tout de même une certaine aisance... Voilà pourquoi on va maintenant en dire quelques mots.

Données structurées

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.

Attention 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.

Reprenons une fois de plus l'exemple du carnet d'adresses. Je sais, c'est un peu comme mes blagues, ça lasse (là, pour ceux qui s'endorment, je signale qu'il y a un jeu de mots), mais c'est encore le meilleur moyen d'avoir un point de comparaison.

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 Contact
  Nom en Caractère * 20
  Prénom en Caractère * 15
  Tel en Caractère * 10
  Mail en Caractère * 20
Fin Structure
```

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

```
Variable Individu en Contact
```

Et si cela me chantait, je pourrais remplir les différentes informations contenues au sein de la variable Individu de la manière suivante :

```
Individu ← "Joker", "Midnight", "0348946532", "allstars@rock.com"
```

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 • "Joker"
Individu.Prénom • "Midnight"
Individu.Tel • "0348946532"
Individu.Mail • "allstars@rockandroll.com"
```

Ainsi, écrire correctement une information dans le fichier est un jeu d'enfant, 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. Et zou !

```
EcrireFichier 3, Individu
```

De la même manière, dans l'autre sens, lorsque j'effectue une opération de lecture dans le fichier Adresses, ma vie en sera considérablement simplifiée : la structure étant faite pour cela, je peux dorénavant me contenter de recopier une ligne du fichier dans une variable de type Contact, et le tour sera joué. Pour charger l'individu suivant du fichier en mémoire vive, il me suffira donc d'écrire :

```
LireFichier 5, Individu
```

Et là, direct, j'ai bien mes quatre renseignements accessibles dans les quatre champs de la variable individu. Tout cela, évidemment, parce que la structure de ma variable Individu correspond parfaitement à la structure des enregistrements de mon fichier. Dans le cas contraire, pour reprendre une expression connue, on ne découpera pas selon les pointillés, et alors, je pense que vous imaginez le carnage...

2. Tableaux de données structurées

Si à partir des types simples, on peut créer des variables et des tableaux de variables, vous me voyez venir, à partir des types structurés, on peut créer des variables structurées... et des tableaux de variables structurées.

Là, bien que pas si difficile que cela, ça commence à devenir un peu compliqué. Parce que 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.

```
Structure Contact
Nom en Caractère * 20
Prénom en Caractère * 15
Tel en Caractère * 10
Mail en Caractère * 20
Fin Structure
```

```
Tableau Mespotes() en Contact
Début
Ouvrir "Exemple.txt" sur 3 en Lecture
i • -1
Tantque Non EOF(3)
   i • i + 1
   Redim Mespotes(i)
   LireFichier 3, Mespotes(i)
FinTantQue
Fermer 3
Fin
```

Une fois que ceci est réglé, on a tout ce qu'il faut ! Si je voulais écrire, à un moment, le mail de l'individu n°13 du fichier (donc le n°12 du tableau) à l'écran, il me suffirait de passer l'ordre :

```
Ecrire Mespotes(12).Mail
```

3. Récapitulatif général

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.

Voici une série de (pas toujours) petits exercices sur les fichiers texte, que l'on pourra traiter en employant les types structurés (c'est en tout cas le cas dans les corrigés).