PARTIE 1
INTRODUCTION A L'ALGORITHMIQUE

L'algorithmique est un terme d'origine arabe, comme algèbre, amiral ou zénith. Ce n'est pas une excuse pour massacrer son orthographe, ou sa prononciation.

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 ?

Avez-vous déjà ouvert un livre de recettes de cuisine ? Avez vous déjà déchiffré un mode d'emploi traduit directement du coréen pour faire fonctionner un magnétoscope ou un répondeur téléphonique réticent ? Si oui, sans le savoir, vous avez déjà exécuté des algorithmes.

Plus fort : avez-vous déjà indiqué un chemin à un touriste égaré ? Avez vous fait chercher un objet à quelqu'un par téléphone ? Ecrit une lettre anonyme stipulant comment procéder à une remise de rançon ? Si oui, vous avez déjà fabriqué – et fait exécuter – des algorithmes.

Comme quoi, l'algorithmique n'est pas un savoir ésotérique réservé à quelques rares initiés touchés par la grâce divine, mais une aptitude partagée par la totalité de l'humanité. Donc, pas d'excuses…

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, et le touriste se retrouve là où il voulait aller. Si l'algorithme est faux, le résultat est, disons, aléatoire, et décidément, cette saloperie de répondeur ne veut rien savoir.

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. Pas de ça Lisette, ce serait trop facile.

Le malheur (ou le bonheur, tout dépend du point de vue) est que justement, si le touriste vous demande son chemin, c'est qu'il ne le connaît pas. Donc, si on n'est pas un goujat intégral, il ne sert à rien de lui dire de le trouver tout seul. De même les modes d'emploi contiennent généralement (mais pas toujours) un peu plus d'informations que « débrouillez vous pour que ça marche ».

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

En informatique, heureusement, il n'y a pas ce problème : les choses auxquelles ont doit donner des instructions sont les ordinateurs, et ceux-ci ont le bon goût d'être tous strictement aussi idiots les uns que les autres.

2. FAUT-IL ETRE MATHEUX POUR ETRE BON EN ALGORITHMIQUE ?

Je consacre quelques lignes à cette question, car cette opinion aussi fortement affirmée que faiblement fondée sert régulièrement d'excuse : « moi, de toute façon, je suis mauvais(e) en algo, j'ai jamais rien pigé aux maths ». Faut-il être « bon en maths » pour expliquer correctement son chemin à quelqu'un ? Je vous laisse juge.

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

- il faut avoir une certaine **intuition**, car aucune recette ne permet de savoir a priori quelles instructions permettront d'obtenir le résultat voulu. C'est là, 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, et j'insiste sur ce point, les réflexes, cela s'acquiert. Et ce qu'on appelle l'intuition n'est finalement que de l'expérience tellement répétée que le raisonnement, au départ laborieux, finit par devenir « spontané ».

- il faut être **méthodique** et **rigoureux**. 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.

Et petit à petit, à force de pratique, vous verrez que vous pourrez faire de plus en plus souvent l'économie de cette dernière étape : l'expérience fera que vous « verrez » le résultat produit par vos instructions, au fur et à mesure que vous les écrirez. Naturellement, cet apprentissage est long, et demande des heures de travail patient. Aussi, dans un premier temps, évitez de sauter les étapes : **la vérification méthodique, pas à pas, de chacun de vos algorithmes représente plus de la moitié du travail à accomplir... et le gage de vos progrès.**

3. L'ADN, LES SHADOKS, ET LES ORDINATEURS

Quel rapport me direz-vous ? Eh bien le point commun est : quatre mots de vocabulaire.

L'univers lexical Shadok, c'est bien connu, se limite aux termes « Ga », « Bu », « Zo », et « Meu ». Ce qui leur a tout de même permis de formuler quelques fortes maximes, telles que : « *Mieux vaut pomper et qu'il ne se passe rien, plutôt qu'arrêter de pomper et risquer qu'il se passe quelque chose de pire* » (pour d'autres fortes maximes Shadok, n'hésitez pas à visiter leur site Internet, il y en a toute une collection qui vaut le détour).

L'ADN, qui est en quelque sorte le programme génétique, l'algorithme à la base de construction des êtres vivants, est une chaîne construite à partir de quatre éléments invariables. Ce n'est que le nombre de ces éléments, ainsi que l'ordre dans lequel ils sont arrangés, qui vont déterminer si on obtient une puce ou un éléphant. Et tous autant que nous sommes, splendides réussites de la Nature, avons été construits par un « programme » constitué uniquement de ces quatre briques, ce qui devrait nous inciter à la modestie.

Enfin, les ordinateurs, quels qu'ils soient, ne sont fondamentalement capables de comprendre que quatre catégories d'ordres (en programmation, on n'emploiera pas le terme d'ordre, mais plutôt celui d'**instructions**). Ces quatre familles d'instructions sont :

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

Un algorithme informatique se ramène donc 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. Rassurez-vous, dans le cadre de ce cours, nous n'irons pas jusque là (cependant, la taille d'un algorithme ne conditionne pas en soi sa complexité : de longs algorithmes peuvent être finalement assez simples, et de petits très compliqués).

4. 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**. 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. Apprendre l'algorithmique de manière séparée, c'est donc sérier les difficultés pour mieux les vaincre.

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 !

Bon, maintenant que j'ai bien fait l'article pour vendre ma marchandise, on va presque pouvoir passer au vif du sujet…

5. 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. Donc, chaque cuisinier peut faire sa sauce à sa guise, avec ses petites épices bien à lui, sans que cela prête à conséquence.

Comme je n'ai pas moins de petites manies que la majorité de mes semblables, le pseudo-code que vous découvrirez dans les pages qui suivent possède quelques spécificités mineures qui ne doivent qu'à mes névroses personnelles.

Rassurez-vous cependant, celles-ci restent dans les limites tout à fait acceptables.

En tout cas, personnellement, je les accepte très bien.

PARTIE 2
LECTURE ET ECRITURE

1. DE QUOI PARLE-T-ON ?

Trifouiller des variables en mémoire vive par un chouette 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 (et Lycée de Versailles, eût ajouté l'estimé Pierre Dac, qui en précurseur méconnu de l'algorithmique, affirmait tout aussi profondément que « *rien ne sert de penser, il faut réfléchir avant* »).

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**.

Remarque essentielle : A première vue, on peut avoir l'impression que les informaticiens étaient beurrés comme des petits lus lorsqu'ils ont baptisé ces opérations ; puisque quand l'utilisateur doit écrire au clavier, on appelle ça la lecture, et quand il doit lire sur l'écran on appelle çà l'écriture. Mais avant d'agonir d'insultes une digne corporation, il faut réfléchir un peu plus loin. Un algorithme, c'est une suite d'instructions qui programme la machine, pas l'utilisateur ! Donc quand on dit à la machine de lire une valeur, cela implique que l'utilisateur va devoir écrire cette valeur. Et quand on demande à la machine d'écrire une valeur, c'est pour que l'utilisateur puisse la lire. Lecture et écriture sont donc des termes qui comme toujours en programmation, doivent être compris du point de vue de la machine qui sera chargée de les exécuter. Et là, tout devient parfaitement logique. Et toc.

2. LES INSTRUCTIONS DE LECTURE ET D'ECRITURE

Tout bêtement, 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).

Et ça y est, vous savez d'ores et déjà sur cette question tout ce qu'il y a à savoir…

PARTIE 3
LES TESTS

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.

Eh bien, croyez le ou non, mais 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**. Toutefois, ceux qui tiennent absolument à briller en société parleront également de **structure alternative**.

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
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**

Nous reviendrons dans quelques instants sur ce qu'est une **condition** en informatique.

Toujours est-il que la structure d'un test est relativement claire. Dans la forme la plus simple, arrivé à la première ligne (Si… Alors) la machine 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 de la structure complète, c'est à peine plus compliqué. Dans le cas où le booléen 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 le booléen 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.

En fait, la forme simplifiée correspond au cas où l'une des deux « branches » du Si est vide. Dès lors, plutôt qu'écrire « sinon ne rien faire du tout », il est plus simple de ne rien écrire. Et laisser un Si... complet, avec une des deux branches vides, est considéré comme une très grosse maladresse pour un programmeur, même si cela ne constitue pas à proprement parler une faute.

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.

A noter que ces opérateurs de comparaison peuvent tout à fait s'employer avec des caractères. Ceux-ci sont codés par la machine dans l'ordre alphabétique (rappelez vous le code ASCII vu dans le préambule), les majuscules étant systématiquement placées avant les minuscules. Ainsi on a :

```
"t" < "w"              VRAI
"Maman" > "Papa"       FAUX
"maman" > "Papa"       VRAI
```

Remarque très importante

En formulant une condition dans un algorithme, il faut se méfier comme de la peste de certains raccourcis du langage courant, ou de certaines notations valides en mathématiques, mais qui mènent à des non-sens informatiques. Prenons par exemple la phrase « Toto est compris entre 5 et 8 ». On peut être tenté de la traduire par : **5 < Toto < 8**

Or, une telle expression, qui a du sens en français, comme en mathématiques, **ne veut rien dire en programmation**. En effet, elle comprend deux opérateurs de comparaison, soit un de trop, et trois valeurs, soit là aussi une de trop. On va voir dans un instant comment traduire convenablement une telle condition.

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.

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

LE GAG DE LA JOURNÉE...

...Consiste à formuler dans un test **une condition qui ne pourra jamais être vraie, ou jamais être fausse**. Si ce n'est pas fait exprès, c'est assez rigolo. Si c'est fait exprès, c'est encore plus drôle, car une condition dont on sait d'avance qu'elle sera toujours fausse n'est pas une condition. Dans tous les cas, cela veut dire qu'on a écrit un test qui n'en est pas un, et qui fonctionne comme s'il n'y en avait pas.
Cela peut être par exemple : Si Toto < 10 ET Toto > 15 Alors… (il est très difficile de trouver un nombre qui soit à la fois inférieur à 10 et supérieur à 15 !)
Bon, ça, c'est un motif immédiat pour payer une tournée générale, et je sens qu'on ne restera pas longtemps le gosier sec.

5. TESTS IMBRIQUES

Graphiquement, on peut très facilement représenter un SI comme un aiguillage de chemin de fer (ou un aiguillage de train électrique, c'est moins lourd à porter). Un SI ouvre donc deux voies, correspondant à deux traitements différents. Mais il y a des tas de situations où deux voies ne suffisent pas. 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 :

```
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.

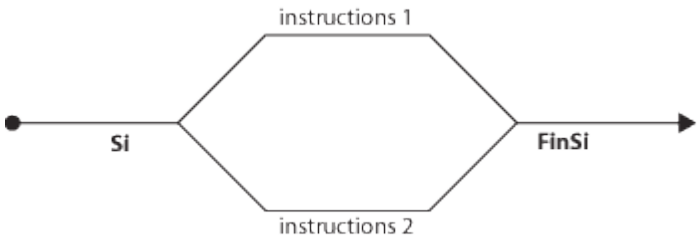
Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

6. De l'aiguillage a la gare de tri

« J'ai l'âme ferroviaire : je regarde passer les vaches » (Léo Ferré)

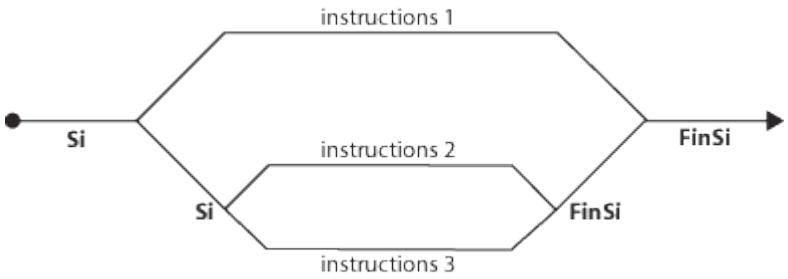
Cette citation n'apporte peut-être pas grand chose à cet exposé, mais je l'aime bien, alors c'était le moment ou jamais.

En effet, dans un programme, une structure SI peut être facilement comparée à un aiguillage de train. La voie principale se sépare en deux, le train devant rouler ou sur l'une, ou sur l'autre, et les deux voies se rejoignant tôt ou tard pour ne plus en former qu'une seule, lors du FinSi. On peut schématiser cela ainsi :



Mais dans certains cas, ce ne sont pas deux voies qu'il nous faut, mais trois, ou même plus. Dans le cas de l'état de l'eau, il nous faut trois voies pour notre « train », puisque l'eau peut être solide, liquide ou gazeuse. Alors, nous n'avons pas eu le choix : pour deux voies, il nous fallait un aiguillage, pour trois voies il nous en faut deux, imbriqués l'un dans l'autre.

Cette structure (telle que nous l'avons programmée à la page précédente) devrait être schématisée comme suit :



Soyons bien clairs : cette structure est la seule possible du point de vue logique (même si on peut toujours mettre le bas en haut et le haut en bas). Mais du point de vue de l'écriture, le pseudo-code algorithmique admet une simplification supplémentaire. Ainsi, il est possible (mais non obligatoire, que l'algorithme initial :

```
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
```

devienne :

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

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

Le **SinonSi** permet en quelque sorte de créer (en réalité, de simuler) des aiguillages à plus de deux branches. On peut ainsi enchaîner les SinonSi les uns derrière les autres pour simuler un aiguillage à autant de branches que l'on souhaite.

7. Variables Booleennes

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).

PARTIE 4
ENCORE DE LA LOGIQUE

1. FAUT-IL METTRE UN ET ? FAUT-IL METTRE UN OU ?

Une remarque pour commencer : dans le cas de conditions composées, les parenthèses jouent un rôle fondamental.

```
Variables A, B, C, D, E en Booléen
Variable X en Entier
Début
Lire X
A ← X > 12
B ← X > 2
C ← X < 6
D ← (A ET B) OU C
E ← A ET (B OU C)
Ecrire D, E
Fin
```

Si X = 3, alors on remarque que D sera VRAI alors que E sera FAUX.

S'il n'y a dans une condition que des ET, ou que des OU, en revanche, les parenthèses ne changent strictement rien.

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.

On en arrive à une autre propriété des ET et des OU, bien plus intéressante.

Spontanément, on pense souvent que ET et OU s'excluent mutuellement, au sens où un problème donné s'exprime soit avec un ET, soit avec un OU. Pourtant, ce n'est pas si évident.

Quand faut-il ouvrir la fenêtre de la salle ? Uniquement si les conditions l'imposent, à savoir :

```
Si il fait trop chaud ET il ne pleut pas Alors
  Ouvrir la fenêtre
Sinon
  Fermer la fenêtre
Finsi
```

Cette petite règle pourrait tout aussi bien être formulée comme suit :

```
Si il ne fait pas trop chaud OU il pleut Alors
  Fermer la fenêtre
Sinon
  Ouvrir la fenêtre
Finsi
```

Ces deux formulations sont strictement équivalentes. Ce qui nous amène à la conclusion suivante :

Toute structure de test requérant une condition composée faisant intervenir l'opérateur ET peut être exprimée de manière équivalente avec un opérateur OU, et réciproquement.

Ceci est moins surprenant qu'il n'y paraît au premier abord. Jetez pour vous en convaincre un œil sur les tables de vérité, et vous noterez la symétrie entre celle du ET et celle du OU. Dans les deux tables, il y a trois cas sur quatre qui mènent à un résultat, et un sur quatre qui mène au résultat inverse. Alors, rien d'étonnant à ce qu'une situation qui s'exprime avec une des tables (un des opérateurs logiques) puisse tout aussi bien être exprimée avec l'autre table (l'autre opérateur logique). Toute l'astuce consiste à savoir effectuer correctement ce passage.

Bien sûr, on ne peut pas se contenter de remplacer purement et simplement les ET par des OU ; ce serait un peu facile. La règle d'équivalence est la suivante (on peut la vérifier sur l'exemple de la fenêtre) :

```
Si A ET B Alors
  Instructions 1
Sinon
  Instructions 2
Finsi
```

```
équivaut à :

Si NON A OU NON B Alors
  Instructions 2
Sinon
  Instructions 1
Finsi
```

Cette règle porte le nom de **transformation de Morgan**, du nom du mathématicien anglais qui l'a formulée.

2. Au-dela de la logique : le style

Ce titre un peu provocateur (mais néanmoins justifié) a pour but d'attirer maintenant votre attention sur un fait fondamental en algorithmique, fait que plusieurs remarques précédentes ont déjà dû vous faire soupçonner : il n'y a jamais une seule manière juste de traiter les structures alternatives. Et plus généralement, il n'y a jamais une seule manière juste de traiter un problème. Entre les différentes possibilités, qui ne sont parfois pas meilleures les unes que les autres, le choix est une affaire de **style**.

C'est pour cela qu'avec l'habitude, on reconnaît le style d'un programmeur aussi sûrement que s'il s'agissait de style littéraire.

Reprenons nos opérateurs de comparaison maintenant familiers, le ET et le OU. En fait, on s'aperçoit que l'on pourrait tout à fait s'en passer ! Par exemple, pour reprendre l'exemple de la fenêtre de la salle :

```
Si il fait trop chaud ET il ne pleut pas Alors
  Ouvrir la fenêtre
Sinon
  Fermer la fenêtre
Finsi
```

Possède un parfait équivalent algorithmique sous la forme de :

```
Si il fait trop chaud Alors
  Si il ne pleut pas Alors
    Ouvrir la fenêtre
  Sinon
    Fermer la fenêtre
  Finsi
Sinon
  Fermer la fenêtre
Finsi
```

Dans cette dernière formulation, nous n'avons plus recours à une condition composée (mais au prix d'un test imbriqué supplémentaire)

Et comme tout ce qui s'exprime par un ET peut aussi être exprimé par un OU, nous en concluons que le OU peut également être remplacé par un test imbriqué supplémentaire. On peut ainsi poser cette règle stylistique générale :

Dans une structure alternative complexe, les conditions composées, l'imbrication des structures de tests et l'emploi des variables booléennes ouvrent la possibilité de choix stylistiques différents. L'alourdissement des conditions allège les structures de tests et le nombre des booléens nécessaires ; l'emploi de booléens supplémentaires permet d'alléger les conditions et les structures de tests, et ainsi de suite.

Si vous avez compris ce qui précède, et que l'exercice de la date ne vous pose plus aucun problème, alors vous savez tout ce qu'il y a à savoir sur les tests pour affronter n'importe quelle situation. Non, ce n'est pas de la démagogie !

Malheureusement, nous ne sommes pas tout à fait au bout de nos peines ; il reste une dernière structure logique à examiner, et pas des moindres…

PARTIE 5
LES BOUCLES

Et ça y est, on y est, on est arrivés, la voilà, c'est Broadway, la quatrième et dernière structure : ça est les **boucles**. Si vous voulez épater vos amis, vous pouvez également parler de **structures répétitives**, voire carrément de **structures itératives**. Ca calme, hein ? Bon, vous faites ce que vous voulez, ici on est entre nous, on parlera de boucles.

Les boucles, c'est généralement le point douloureux de l'apprenti programmeur. C'est là que ça coince, car autant il est assez facile de comprendre comment fonctionnent les boucles, autant il est souvent long d'acquérir les réflexes qui permettent de les élaborer judicieusement pour traiter un problème donné.

On peut dire en fait que les boucles constituent la seule vraie structure logique caractéristique de la programmation. Si vous avez utilisé un tableur comme Excel, par exemple, vous avez sans doute pu manier des choses équivalentes aux variables (les cellules, les formules) et aux tests (la fonction SI…). Mais les boucles, ça, ça n'a aucun équivalent. Cela n'existe que dans les langages de programmation proprement dits.

Le maniement des boucles, s'il ne différencie certes pas l'homme de la bête (il ne faut tout de même pas exagérer), est tout de même ce qui sépare en informatique le programmeur de l'utilisateur, même averti.

Alors, à vos futures – et inévitables - difficultés sur le sujet, il y a trois remèdes : de la rigueur, de la patience, et encore de la rigueur !

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.

A vue de nez, 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, on peut rajouter des centaines de SI, et écrire un algorithme aussi lourd qu'une blague des Grosses Têtes, on n'en sortira pas, il y aura toujours moyen qu'un acharné flanque le programme par terre.

La solution consistant à aligner des SI… en pagaille est donc une impasse. La seule issue est donc de flanquer 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 (qui, je le rappelle, peut être une variable booléenne ou, plus fréquemment, une 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. Le manège enchanté ne s'arrête que lorsque le booléen prend la valeur FAUX.

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"
  Lire Rep
FinTantQue
Fin
```

Là, on a le squelette de l'algorithme correct. Mais de même qu'un squelette ne suffit pas pour avoir un être vivant viable, il va nous falloir ajouter quelques muscles et organes sur cet algorithme pour qu'il fonctionne correctement.

Son principal défaut est de provoquer une erreur à chaque exécution. En effet, l'expression booléenne qui figure après le TantQue interroge la valeur de la variable Rep. Malheureusement, cette variable, si elle a été déclarée, n'a pas été affectée avant l'entrée dans la boucle. On teste donc une variable qui n'a pas de valeur, ce qui provoque une erreur et l'arrêt immédiat de l'exécution. Pour éviter ceci, on n'a pas le choix : il faut que la variable Rep ait déjà été affectée avant qu'on en arrive au premier tour de boucle. Pour cela, on peut faire une première lecture de Rep avant la boucle. Dans ce cas, celle-ci ne servira qu'en cas de mauvaise saisie lors de cette première lecture. L'algorithme devient alors :

```
Variable Rep en Caractère
Début
Ecrire "Voulez vous un café ? (O/N)"
Lire Rep
TantQue Rep <> "O" et Rep <> "N"
  Lire Rep
FinTantQue
Fin
```

Une autre possibilité, fréquemment employée, consiste à ne pas lire, mais à affecter arbitrairement la variable avant la boucle. Arbitrairement ? Pas tout à fait, puisque cette affectation doit avoir pour résultat de provoquer l'entrée obligatoire dans la boucle. L'affectation doit donc faire en sorte que le booléen soit mis à VRAI pour déclencher le premier tour de la boucle. Dans notre exemple, on peut donc affecter Rep avec n'importe quelle valeur, hormis « O » et « N » : car dans ce cas, l'exécution sauterait la boucle, et Rep ne serait pas du tout lue au clavier. Cela donnera par exemple :

```
Variable Rep en Caractère
Début
Rep ← "X"
Ecrire "Voulez vous un café ? (O/N)"
TantQue Rep <> "O" et Rep <> "N"
  Lire Rep
FinTantQue
Fin
```

Cette manière de procéder est à connaître, car elle est employée très fréquemment.

Il faut remarquer que les deux solutions (lecture initiale de Rep en dehors de la boucle ou affectation de Rep) rendent toutes deux l'algorithme satisfaisant, mais présentent une différence assez importante dans leur structure logique.

En effet, si l'on choisit d'effectuer une lecture préalable de Rep, la boucle ultérieure sera exécutée uniquement dans l'hypothèse d'une mauvaise saisie initiale. Si l'utilisateur saisit une valeur correcte à la première demande de Rep, l'algorithme passera sur la boucle sans entrer dedans.

En revanche, avec la deuxième solution (celle d'une affectation préalable de Rep), l'entrée de la boucle est forcée, et l'exécution de celle-ci, au moins une fois, est rendue obligatoire à chaque exécution du programme. Du point de vue de l'utilisateur, cette différence est tout à fait mineure ; et à la limite, il ne la remarquera même pas. Mais du point de vue du programmeur, il importe de bien comprendre que les cheminements des instructions ne seront pas les mêmes dans un cas et dans l'autre.

Pour terminer, remarquons que nous pourrions peaufiner nos solutions en ajoutant des affichages de libellés qui font encore un peu défaut. Ainsi, si l'on est un programmeur zélé, la première solution (celle qui inclut deux lectures de Rep, une en dehors de la boucle, l'autre à l'intérieur) pourrait devenir :

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

Quant à la deuxième solution, elle pourra devenir :

```
Variable Rep en Caractère
Début
Rep ← "X"
Ecrire "Voulez vous un café ? (O/N)"
TantQue Rep <> "O" et Rep <> "N"
  Lire Rep
  Si Rep <> "O" et Rep <> "N" Alors
    Ecrire "Saisie Erronée, Recommencez"
  FinSi
FinTantQue
Fin
```

Le Gag De La Journée

C'est d'écrire une structure TantQue dans laquelle le booléen n'est jamais VRAI. Le programme ne rentre alors jamais dans la superbe boucle sur laquelle vous avez tant sué !

Mais la faute symétrique est au moins aussi désopilante.

Elle consiste à écrire une boucle dans laquelle le booléen ne devient jamais FAUX. L'ordinateur tourne alors dans la boucle comme un dératé et n'en sort plus. Seule solution, quitter le programme avec un démonte-pneu ou un bâton de dynamite. La « **boucle infinie** » est une des hantises les plus redoutées des programmeurs. C'est un peu comme le verre baveur, le poil à gratter ou le bleu de méthylène : c'est éculé, mais ça fait toujours rire.

Cette faute de programmation grossière – mais fréquente - ne manquera pas d'égayer l'ambiance collective de cette formation… et accessoirement d'étancher la soif proverbiale de vos enseignants.

Bon, eh bien vous allez pouvoir faire de chouettes algorithmes, déjà rien qu'avec ça…

2. BOUCLER EN COMPTANT, OU COMPTER EN BOUCLANT

Dans le dernier exercice, vous avez remarqué qu'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 :

```
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**, encore un mot qui fera forte impression sur votre entourage).

Dit d'une autre manière, la structure « Pour … Suivant » 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 … 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 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 (Le « pas » dont nous parlons, c'est le « pas » du marcheur, « step » en anglais).

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.

Nous pouvons donc maintenant donner la formulation générale d'une structure « Pour ». Sa syntaxe générale est :

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

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(cf. Partie 9)

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.

Nous verrons dans les chapitres suivants des séries d'éléments appelés tableaux (parties 7 et 8) et chaînes de caractères (partie 9). Selon les cas, le balayage systématique des éléments de ces séries pourra être effectué par un Pour ou par un TantQue : tout dépend si la quantité d'éléments à balayer (donc le nombre de tours de boucles nécessaires) peut être dénombrée à l'avance par le programmeur ou non.

3. DES BOUCLES DANS DES BOUCLES (« *TOUT EST DANS TOUT... ET RECIPROQUEMENT* »)

On rigole, on rigole !

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. 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. Cela n'aurait aucun sens logique, et de plus, bien peu de langages vous autoriseraient ne serait-ce qu'à écrire cette structure aberrante.

```
Variables Truc, Trac en Entier
Pour Truc ← …
  instructions
  Pour Trac ← …
    instructions
Truc Suivant
  instructions
  Trac Suivant
```

Pourquoi imbriquer des boucles ? Pour la même raison qu'on imbrique des tests. La traduction en bon français d'un test, c'est un « cas ». Eh bien un « cas » (par exemple, « est-ce un homme ou une femme ? ») peut très bien se subdiviser en d'autres cas (« a-t-il plus ou moins de 18 ans ? »).

De même, une boucle, c'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).

Dans la pratique de la programmation, la maîtrise des boucles imbriquées est nécessaire, même si elle n'est pas suffisante. Tout le contraire d'Alain Delon, en quelque sorte.

4. ET ENCORE UNE BETISE A NE PAS FAIRE !

Examinons l'algorithme suivant :

```
Variable Truc en Entier
Début
Pour Truc ← 1 à 15
  Truc ← Truc * 2
  Ecrire "Passage numéro : ", Truc
Truc Suivant
Fin
```

Vous remarquerez que nous faisons ici gérer « en double » la variable Truc, ces deux gestions étant contradictoires. D'une part, la ligne

```
Pour…
```

augmente la valeur de Truc de 1 à chaque passage. D'autre part la ligne

```
Truc ← Truc * 2
```

double la valeur de Truc à chaque passage. Il va sans dire que de telles manipulations perturbent complètement le déroulement normal de la boucle, et sont causes, sinon de plantages, tout au moins d'exécutions erratiques.

Le Gag De La Journée

Il consiste donc à manipuler, au sein d'une boucle **Pour**, la variable qui sert de compteur à cette boucle. Cette technique est à proscrire absolument… sauf bien sûr, si vous cherchez un prétexte pour régaler tout le monde au bistrot.

Mais dans ce cas, n'ayez aucune inhibition, proposez-le directement, pas besoin de prétexte.

PARTIE 6
LES TABLEAUX

Bonne nouvelle ! Je vous avais annoncé qu'il y a avait en tout et pour tout quatre structures logiques dans la programmation. Eh bien, ça y est, on les a toutes passées en revue.

Mauvaise nouvelle, il vous reste tout de même quelques petites choses à apprendre…

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 atrocité du genre :

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

Ouf ! C'est tout de même bigrement 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 le suicide direct.

Cerise sur le gâteau, si en plus on est dans une situation on l'on ne peut pas savoir d'avance combien il y aura de valeurs à traiter, là on est carrément cuits.

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 - ô surprise - 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.

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. Eh oui, 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). Au début, ça déroute, mais vous verrez, avec le temps, on se fait à tout, même au pire.

```
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
```

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 0** (dans quelques rares langages, le premier élément d'un tableau porte l'indice 1). Mais comme je l'ai déjà écrit plus haut, nous avons choisi ici de commencer la numérotation des indices à zéro, comme c'est le cas en langage C et en Visual Basic. Donc attention, Truc(6) est le septième élément du tableau Truc !
- **être un nombre entier** Quel que soit le langage, l'élément Truc(3,1416) n'existe jamais.
- **être inférieure ou égale au nombre d'éléments du tableau** (moins 1, si l'on commence la numérotation à zéro). Si le tableau Bidule a été déclaré comme ayant 25 éléments, la présence dans une ligne, sous une forme ou sous une autre, de Bidule(32) déclenchera automatiquement une erreur.

Je le re-re-répète, si l'on est dans un langage où les indices commencent à zéro, il faut en tenir compte à la déclaration :

```
Tableau Note(13) en Numérique
```

...créera un tableau de 14 éléments, le plus petit indice étant 0 et le plus grand 13.

LE GAG DE LA JOURNEE

Il consiste à confondre, dans sa tête et / ou dans un algorithme, l'**indice** d'un élément d'un tableau avec le **contenu** de cet élément. La troisième maison de la rue n'a pas forcément trois habitants, et la vingtième vingt habitants. En notation algorithmique, il n'y a aucun rapport entre i et truc(i).

Holà, Tavernier, prépare la cervoise !

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 les varices) 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)
…
```

Cette technique n'a rien de sorcier, mais elle fait partie de l'arsenal de base de la programmation en gestion.

PARTIE 7
TECHNIQUES RUSEES

Une fois n'est pas coutume, ce chapitre n'a pas pour but de présenter un nouveau type de données, un nouveau jeu d'instructions, ou que sais-je encore.

Son propos est de détailler quelques techniques de programmation qui possèdent deux grands points communs :

- leur connaissance est parfaitement indispensable
- elles sont un rien finaudes

Et que valent quelques kilos d'aspirine, comparés à l'ineffable bonheur procuré par la compréhension suprême des arcanes de l'algorithmique ? Hein ?

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

Première de ces ruses de sioux, et par ailleurs tarte à la crème absolue du programmeur, donc : le tri de tableau.

Combien de fois au cours d'une carrière (brillante) de développeur a-t-on besoin de ranger des valeurs dans un ordre donné ? C'est inimaginable. Aussi, plutôt qu'avoir à réinventer à chaque fois la roue, le fusil à tirer dans les coins, le fil à couper le roquefort et la poudre à maquiller, vaut-il mieux avoir assimilé une ou deux techniques solidement éprouvées, même si elles paraissent un peu ardues au départ.

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. Champagne !

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 in fine :

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

Et cetera, et cetera, jusqu'à l'avant dernier.

En bon français, 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 :

```
boucle principale : le point de départ se décale à chaque tour
Pour i ← 0 à 10
on considère provisoirement que t(i) est le plus petit élément
  posmini ← i
on examine tous les éléments suivants
  Pour j ← i + 1 à 11
    Si t(j) < t(posmini) Alors
       posmini ← j
    Finsi
  j suivant
A cet endroit, on sait maintenant où est le plus petit élément. Il ne reste plus qu'à effectuer
la permutation.
  temp ← t(posmini)
  t(posmini) ← t(i)
  t(i) ← temp
On a placé correctement l'élément numéro i, on passe à présent au suivant.
i suivant
```

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.

Tout ceci peut vous sembler un peu fumeux, mais cela devrait s'éclairer à l'aide d'un exemple extrêmement fréquent : la recherche de l'occurrence d'une valeur dans un tableau. On en profitera au passage pour corriger une erreur particulièrement fréquente chez le programmeur débutant.

Soit un tableau comportant, disons, 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 – car il y en a manifestement une – de parcours du tableau :

```
Tableau Tab(19) en Numérique
Variable N en Numérique
Début
Ecrire "Entrez la valeur à rechercher"
Lire N
Pour i ← 0 à 19
???
```

```
i suivant
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 bingo, N fait partie du tableau. Cela va se traduire, bien entendu, par un Si … Alors … Sinon. Et voilà le programmeur raisonnant hâtivement qui se vautre en écrivant :

```
Tableau Tab(19) en Numérique
Variable N en Numérique
Début
Ecrire "Entrez la valeur à rechercher"
Lire N
Pour i ← 0 à 19
  Si N = Tab(i) Alors
    Ecrire "N fait partie du tableau"
  Sinon
    Ecrire "N ne fait pas partie du tableau"
  FinSi
i suivant
Fin
```

Et patatras, cet algorithme est une véritable catastrophe.

Il suffit d'ailleurs de le faire tourner mentalement pour s'en rendre compte. De deux choses l'une : ou bien la valeur N figure dans le tableau, ou bien elle n'y figure pas. Mais dans tous les cas, **l'algorithme ne doit produire qu'une seule réponse, quel que soit le nombre d'éléments que compte le tableau. Or, l'algorithme ci-dessus envoie à l'écran autant de messages qu'il y a de valeurs dans le tableau**, en l'occurrence pas moins de 20 !

Il y a donc une erreur manifeste de conception : l'écriture du message ne peut se trouver à l'intérieur de la boucle : elle doit figurer à l'extérieur. On sait si la valeur était dans le tableau ou non **uniquement lorsque le balayage du tableau est entièrement accompli**.

Nous réécrivons donc cet algorithme en plaçant le test après la boucle. Faute de mieux, on se contentera de faire dépendre pour le moment la réponse d'une variable booléenne que nous appellerons Trouvé.

```
Tableau Tab(19) en Numérique
Variable N en Numérique
Début
Ecrire "Entrez la valeur à rechercher"
Lire N
Pour i ← 0 à 19
   ???
i suivant
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). Et attention : le test est asymétrique. Il ne comporte pas de "sinon". On reviendra là dessus dans un instant.
- last, but not least, l'affectation par défaut de la variable Trouvé, dont la valeur de départ doit être évidemment Faux.

Au total, l'algorithme complet – et juste ! – donne :

```
Tableau Tab(19) en Numérique
Variable N en Numérique
```

```
Début
Ecrire "Entrez la valeur à rechercher"
Lire N
Trouvé ← Faux
Pour i ← 0 à 19
  Si N = Tab(i) Alors
    Trouvé ← Vrai
  FinSi
i suivant
Si Trouvé Alors
  Ecrire "N fait partie du tableau"
Sinon
  Ecrire "N ne fait pas partie du tableau"
FinSi
Fin
```

Méditons un peu sur cette affaire.

La difficulté est de comprendre que dans une recherche, le problème ne se formule pas de la même manière selon qu'on le prend par un bout ou par un autre. On peut résumer l'affaire ainsi : **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 **, un « drapeau » qui peut se lever, mais jamais se rabaisser**. Et cette technique de flag (que nous pourrions élégamment surnommer « gestion asymétrique de variable booléenne ») doit être mise en œuvre chaque fois que l'on se trouve devant pareille situation.

Autrement dit, connaître ce type de raisonnement est indispensable, et savoir le reproduire à bon escient ne l'est pas moins.

3. TRI DE TABLEAU + FLAG = TRI A BULLES

Et maintenant, nous en arrivons à la formule magique : tri de tableau + flag = tri à 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**. Cette constatation percutante semble digne de M. de Lapalisse, un ancien voisin à moi. Mais elle est plus profonde – et plus utile - qu'elle n'en a l'air.

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 ». Comme quoi l'algorithmique n'exclut pas un minimum syndical de sens poétique.

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 :

```
Variable Yapermute en Booléen
Début
…
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. C'est parti :

```
Variable Yapermute en Booléen
Début
…
TantQue Yapermute
Pour i ← 0 à 10
  Si t(i) > t(i+1) Alors
    temp ← t(i)
    t(i) ← t(i+1)
    t(i+1) ← temp
  Finsi
i suivant
Fin
```

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 :

```
Variable Yapermute en Booléen
Début
…
Yapermut ← Vrai
TantQue Yapermut
  Yapermut ← Faux
  Pour i ← 0 à 10
    Si t(i) > t(i+1) alors
      temp ← t(i)
      t(i) ← t(i+1)
      t(i+1) ← temp
      Yapermut ← Vrai
    Finsi
  i suivant
FinTantQue
Fin
```

Au risque de me répéter, la compréhension et la maîtrise du principe du flag font partie de l'arsenal du programmeur bien armé.

4. LA RECHERCHE DICHOTOMIQUE

Je ne sais pas si on progresse vraiment en algorithmique, mais en tout cas, qu'est-ce qu'on apprend comme vocabulaire !

Blague dans le coin, nous allons terminer ce chapitre migraineux par une technique célèbre de recherche, qui révèle toute son utilité lorsque le nombre d'éléments est très élevé. Par exemple, 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 la louche, 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 pile poil au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans le première moitié du dico. 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 ! Il n'y a pas photo sur l'écart de performances entre la technique barbare et la technique futée. Attention, toutefois, même si c'est évident, je le répète avec force : la recherche dichotomique ne peut s'effectuer que sur des éléments préalablement triés.

Eh bien maintenant que je vous ai expliqué comment faire, vous n'avez plus qu'à traduire !

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

PARTIE 8
TABLEAUX MULTIDIMENSIONNELS

Ceci n'est pas un dérèglement de votre téléviseur. Nous contrôlons tout, nous savons tout, et les phénomènes paranormaux que vous constatez sont dus au fait que vous êtes passés dans… la quatrième dimension (musique angoissante : « tintintin… »).

Oui, enfin bon, avant d'attaquer la quatrième, on va déjà se coltiner la deuxième.

1. POURQUOI PLUSIEURS DIMENSIONS ?

Une seule ne suffisait-elle pas déjà amplement à notre bonheur, me demanderez-vous ? Certes, répondrai-je, mais vous allez voir qu'avec deux (et davantage encore) c'est carrément le nirvana.

Prenons le cas de la modélisation d'un jeu de dames, et du déplacement des pions sur le damier. Je rappelle qu'un pion qui est sur une case blanche peut se déplacer (pour simplifier) sur les quatre cases blanches adjacentes.

Avec les outils que nous avons abordés jusque là, le plus simple serait évidemment de modéliser le damier sous la forme d'un tableau. Chaque case est un emplacement du tableau, qui contient par exemple 0 si elle est vide, et 1 s'il y a un pion. On attribue comme indices aux cases les numéros 1 à 8 pour la première ligne, 9 à 16 pour la deuxième ligne, et ainsi de suite jusqu'à 64.

Arrivés à ce stade, les fines mouches du genre de Cyprien L. m'écriront pour faire remarquer qu'un damier, cela possède 100 cases et non 64, et qu'entre les damiers et les échiquiers, je me suis joyeusement emmêlé les pédales. A ces fines mouches, je ferai une double réponse de prof :

1. C'était pour voir si vous suiviez.

2. Si le prof décide contre toute évidence que les damiers font 64 cases, c'est le prof qui a raison et l'évidence qui a tort. Rompez.

Reprenons. Un pion placé dans la case numéro i, autrement dit la valeur 1 de Cases(i), peut bouger vers les cases contiguës en diagonale. Cela va nous obliger à de petites acrobaties intellectuelles : la case située juste au-dessus de la case numéro i ayant comme indice i-8, les cases valables sont celles d'indice i-7 et i-9. De même, la case située juste en dessous ayant comme indice i+8, les cases valables sont celles d'indice i+7 et i+9.

Bien sûr, on peut fabriquer tout un programme comme cela, mais le moins qu'on puisse dire est que cela ne facilite pas la clarté de l'algorithme.

Il serait évidemment plus simple de modéliser un damier par… un damier !

2. 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 Cases(7, 7) en Numérique
```

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).

Pour notre problème de dames, les choses vont sérieusement s'éclaircir. La case qui contient le pion est dorénavant Cases(i, j). Et les quatre cases disponibles sont Cases(i-1, j-1), Cases(i-1, j+1), Cases(i+1, j-1) et Cases(i+1, j+1).

REMARQUE ESSENTIELLE :

Il n'y a aucune différence qualitative entre un tableau à deux dimensions (i, j) et un tableau à une dimension (i * j). De même que le jeu de dames qu'on vient d'évoquer, tout problème qui peut être modélisé d'une manière peut aussi être modélisé de l'autre. Simplement, l'une ou l'autre de ces techniques correspond plus

spontanément à tel ou tel problème, et facilite donc (ou complique, si on a choisi la mauvaise option) l'écriture et la lisibilité de l'algorithme.

Une autre remarque : une question classique à propos des tableaux à deux dimensions est de savoir si le premier indice représente les lignes ou le deuxième les colonnes, ou l'inverse. Je ne répondrai pas à cette question non parce que j'ai décidé de bouder, mais **parce qu'elle n'a aucun sens**. « Lignes » et « Colonnes » sont des concepts graphiques, visuels, qui s'appliquent à des objets du monde réel ; les indices des tableaux ne sont que des coordonnées logiques, pointant sur des adresses de mémoire vive. Si cela ne vous convainc pas, pensez à un jeu de bataille navale classique : les lettres doivent-elles désigner les lignes et les chiffres les colonnes ? Aucune importance ! Chaque joueur peut même choisir une convention différente, aucune importance ! L'essentiel est qu'une fois une convention choisie, un joueur conserve la même tout au long de la partie, bien entendu.

3. 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.

Le principal obstacle au maniement systématique de ces tableaux à plus de trois dimensions est que le programmeur, quand il conçoit son algorithme, aime bien faire des petits gribouillis, des dessins immondes, imaginer les boucles dans sa tête, etc. Or, autant il est facile d'imaginer concrètement un tableau à une dimension, autant cela reste faisable pour deux dimensions, autant cela devient l'apanage d'une minorité privilégiée pour les tableaux à trois dimensions (je n'en fais malheureusement pas partie) et hors de portée de tout mortel au-delà. C'est comme ça, l'esprit humain a du mal à se représenter les choses dans l'espace, et crie grâce dès qu'il saute dans l'hyperespace (oui, c'est comme ça que ça s'appelle au delà de trois dimensions).

Donc, pour des raisons uniquement pratiques, les tableaux à plus de trois dimensions sont rarement utilisés par des programmeurs non matheux (car les matheux, de par leur formation, ont une fâcheuse propension à manier des espaces à n dimensions comme qui rigole, mais ce sont bien les seuls, et laissons les dans leur coin, c'est pas des gens comme nous).

PARTIE 9
LES FONCTIONS PREDEFINIES

Certains traitements ne peuvent être effectués par un algorithme, aussi savant soit-il. D'autres ne peuvent l'être qu'au prix de souffrances indicibles.

C'est par exemple le cas du calcul du sinus d'un angle : pour en obtenir une valeur approchée, il faudrait appliquer une formule d'une complexité à vous glacer le sang. Aussi, que se passe-t-il sur les petites calculatrices que vous connaissez tous ? On vous fournit quelques touches spéciales, dites **touches de fonctions**, qui vous permettent par exemple de connaître immédiatement ce résultat. Sur votre calculatrice, si vous voulez connaître le sinus de 35°, vous taperez 35, puis la touche SIN, et vous aurez 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.

1. STRUCTURE GENERALE DES FONCTIONS

Reprenons l'exemple du sinus. Les langages informatiques, qui se doivent tout de même de savoir faire la même chose qu'une calculatrice à 19F90, proposent généralement une fonction SIN. Si nous voulons stocker le sinus de 35 dans la variable A, nous écrirons :

```
A ← Sin(35)
```

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. 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 ne s'invente pas : il est fixé par le langage. Par exemple, la fonction sinus a besoin d'un argument (ce n'est pas surprenant, cet argument est la valeur de l'angle). Si vous essayez de l'exécuter en lui donnant deux arguments, ou aucun, cela déclenchera une erreur à l'exécution. Notez également que les arguments doivent être d'un certain **type**, et qu'il faut respecter ces types.

Et d'entrée, nous trouvons :

LE GAG DE LA JOURNEE

Il consiste à affecter une fonction, quelle qu'elle soit.

Toute écriture plaçant une fonction à gauche d'une instruction d'affectation est aberrante, pour deux raisons symétriques.

- d'une part, parce que nous le savons depuis le premier chapitre de ce cours extraordinaire, on ne peut affecter qu'une variable, à l'exclusion de tout autre chose.
- ensuite, parce qu'une fonction a pour rôle de produire, de renvoyer, de valoir (tout cela est synonyme), un résultat. Pas d'en recevoir un, donc.

L'affectation d'une fonction sera donc considérée comme l'une des pires fautes algorithmiques, et punie comme telle.

Tavernier...

2. LES FONCTIONS DE TEXTE

Une catégorie privilégiée 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 (moins douloureuses, il faut le noter, que les extractions dentaires).

Tous les langages, je dis bien tous, proposent peu ou prou les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

- **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.

Exemples :

```
Len("Bonjour, ça va ?")                vaut      16
Len("")                                vaut      0
Mid("Zorro is back", 4, 7)             vaut      "ro is b"
Mid("Zorro is back", 12, 1)            vaut      "c"
Left("Et pourtant…", 8)                vaut      "Et pourt"
Right("Et pourtant…", 4)               vaut      "t…"
Trouve("Un pur bonheur", "pur")        vaut       4
Trouve("Un pur bonheur", "techno")     vaut       0
```

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

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

J'insiste ; à moins de programmer avec un langage un peu particulier, comme le C, qui traite en réalité les chaînes de caractères comme des tableaux, on ne pourrait pas se passer des deux fonctions Len et Mid pour traiter les chaînes. Or, si les programmes informatiques ont fréquemment à traiter des nombres, ils doivent tout aussi fréquemment gérer des séries de caractères (des chaînes). Je sais bien que cela devient un refrain, mais connaître les techniques de base sur les chaînes est plus qu'utile : c'est indispensable.

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
```

Cette fonction est notamment indispensable pour effectuer le célébrissime test de parité (voir exercice dans pas longtemps).

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
B ← Mod(12,2)                B vaut 0 car 12 = 6*2
C ← Mod(44,8)                C vaut 4 car 44 = 5*8 + 4
```

Cette fonction peut paraître un peu bizarre, est réservée aux seuls matheux. Mais vous aurez là aussi l'occasion de voir dans les exercices à venir que ce n'est pas le cas.

Génération de nombres aléatoires

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

Tous les programmes de jeu, ou presque, ont besoin de ce type d'outils, qu'il s'agisse de simuler un lancer de dés ou le déplacement chaotique du vaisseau spatial de l'enfer de la mort piloté par l'infâme Zorglub, qui veut faire main basse sur l'Univers (heureusement vous êtes là pour l'en empêcher, ouf).

Mais il n'y a pas que les jeux qui ont besoin de générer des nombres aléatoires. La modélisation (physique, géographique, économique, etc.) a parfois recours à des modèles dits stochastiques (chouette, encore un nouveau mot savant !). Ce sont des modèles dans lesquels les variables se déduisent les unes des autres par des relations déterministes (autrement dit des calculs), mais où l'on simule la part d'incertitude par une « fourchette » de hasard.

Par exemple, un modèle démographique supposera qu'une femme a en moyenne x enfants au cours de sa vie, mettons 1,5. Mais il supposera aussi que sur une population donnée, ce chiffre peut fluctuer entre 1,35 et 1,65 (si on laisse une part d'incertitude de 10%). Chaque année, c'est-à-dire chaque série de calcul des valeurs du modèle, on aura ainsi besoin de faire choisir à la machine un nombre au hasard compris entre 1,35 et 1,65.

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. Je sais bien que mes lecteurs ne sont guère matheux, mais là, on reste franchement en deçà du niveau de feu le BEPC :

- 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. Et le tour est joué !

Bon, il est grand temps que vous montriez ce que vous avez appris…

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.

Rappelez-vous ce que nous avons vu dans les premières pages de ce cours : il existe différents types de variables, qui déterminent notamment le type de codage qui sera utilisé. Prenons le chiffre 3. Si je le stocke dans une variable de type alphanumérique, il sera codé en tant que caractère, sur un octet. Si en revanche je le stocke dans une variable de type entier, il sera codé sur deux octets. Et la configuration des bits sera complètement différente dans les deux cas.

Une conclusion évidente, et sur laquelle on a déjà eu l'occasion d'insister, c'est qu'on ne peut pas faire n'importe quoi avec n'importe quoi, et qu'on ne peut pas par exemple multiplier "3" et "5", si 3 et 5 sont stockés dans des

variables de type caractère. Jusque là, pas de scoop me direz-vous, à juste titre vous répondrai-je, mais attendez donc la suite.

Pourquoi ne pas en tirer les conséquences, et stocker convenablement les nombres dans des variables numériques, les caractères dans des variables alphanumériques, comme nous l'avons toujours fait ?

Parce qu'il y a des situations où on n'a pas le choix ! Nous allons voir dès le chapitre suivant un mode de stockage (les fichiers textes) où toutes les informations, quelles qu'elles soient, sont obligatoirement stockées sous forme de caractères. Dès lors, si l'on veut pouvoir récupérer des nombres et faire des opérations dessus, il va bien falloir être capable de convertir ces chaînes en numériques.

Aussi, tous les langages proposent-ils une palette de fonctions destinées à opérer de telles 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).

PARTIE 10
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

Vous connaissez tous le coup des papous : « *chez les papous, il y a les papous papas et les papous pas papas. Chez les papous papas, il y a les papous papas à poux et les papous papas pas à poux, etc.* » Eh bien les fichiers, c'est un peu pareil : il y a des catégories, et dans les catégories, des sortes, et dans les sortes des espèces. Essayons donc de débroussailler un peu tout cela...

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'il 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 exact 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 sîr 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'ACCES

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 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. Pour des informations plus complètes sur la gestion des fichiers binaires et des autres types d'accès, il vous faudra... chercher ailleurs.

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, 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 rabord, 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
Ouvrir "Exemple.txt" sur 5 en Lecture
Début
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. 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** (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… Ouf.

- 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.

6. DONNEES STRUCTUREES

6.1 Données structurées simples

Nostalgiques du Lego, cette partie va vous plaire. Comment construire des trucs pas possibles et des machins pas croyables avec juste quelques éléments de base ? Vous n'allez pas tarder à le savoir...

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).

Eh bien toujours plus haut, toujours plus fort, 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 Bottin
  Nom en Caractère * 20
  Prénom en Caractère * 15
  Tel en Caractère * 10
  Mail en Caractère * 20
Fin Structure
```

Ici, Bottin 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 Bottin
```

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 Bottin, 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...

6.2 Tableaux de données structurées

Et encore plus loin, encore plus vite et encore plus fort. 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 vraiment balèze. 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 tableaux 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 Bottin
Nom en Caractère * 20
Prénom en Caractère * 15
Tel en Caractère * 10
Mail en Caractère * 20
Fin Structure
Tableau Mespotes() en Bottin
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
```

Et voilà le travail. Simplissime, non ?

REMARQUE FINALE SUR LES DONNÉES STRUCTURÉES

Même si le domaine de prédilection des données structurées est la gestion de fichiers, on peut tout à fait y avoir recours dans d'autres contextes, et organiser plus systématiquement les variables d'un programme sous la forme de telles structures.

En programmation dite **procédurale**, celle que nous étudions ici, ce type de stratégie reste relativement rare. Mais rare ne veut pas dire interdit, ou même inutile.

Et nous aurons l'occasion de voir qu'en programmation **objet**, ce type d'organisation des données devient fondamental.

Mais ceci est un autre cours...

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.

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).

Et en conclusion de la conclusion, voilà plusieurs remarques fondamentales :

REMARQUE N°1

Lorsqu'on veut récupérer des données numériques inscrites dans un fichier texte, il ne faut surtout pas oublier que ces données se présentent forcément sous forme de caractères. La récupération elle-même transmettra donc obligatoirement des données de type alphanumérique ; pour utiliser ces données à des fins ultérieures de calcul, il sera donc nécessaire d'employer une fonction de conversion.

Cette remarque ne s'applique évidemment pas aux fichiers binaires.

REMARQUE N°1bis

Voilà pourquoi une structure s'appliquant aux fichiers textes est forcément composée **uniquement de types caractères**. Une structure traitant de fichiers binaires pourrait en revanche être composée de caractères, de numériques et de booléens.

REMARQUE N°2

Plusieurs langages interdisent l'écriture d'une variable structurée dans un fichier texte, ne l'autorisant que pour un fichier binaire.

Si l'on se trouve dans ce cas, cela signifie qu'on peut certes utiliser une structure, ou un tableau de structures, mais à condition d'écrire sur le fichier champ par champ, ce qui annule une partie du bénéfice de la structure.

Nous avons postulé ici que cette interdiction n'existait pas ; en tenir compte ne changerait pas fondamentalement les algorithmes, mais alourdirait un peu le code pour les lignes traitant de l'écriture dans les fichiers.

PARTIE 11
PROCEDURES ET FONCTIONS

1. FONCTIONS PERSONNALISEES

1.1 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.

La manière la plus évidente, mais aussi la moins habile, de programmer ce genre de choses, c'est bien entendu de répéter le code correspondant autant de fois que nécessaire. Apparemment, on ne se casse pas la tête : quand il faut que la machine interroge l'utilisateur, on recopie les lignes de codes voulues en ne changeant que le nécessaire, et roule Raoul. Mais en procédant de cette manière, la pire qui soit, on se prépare des lendemains qui déchantent...

D'abord, parce que si la structure d'un programme écrit de cette manière peut paraître simple, elle est en réalité inutilement lourdingue. Elle contient des répétitions, et pour peu que le programme soit joufflu, il peut devenir parfaitement illisible. Or, le fait d'être facilement modifiable donc lisible, y compris - et surtout - par ceux qui ne l'ont pas écrit est un critère essentiel pour un programme informatique ! Dès que l'on programme non pour soi-même, mais dans le cadre d'une organisation (entreprise ou autre), cette nécessité se fait sentir de manière aiguë. L'ignorer, c'est donc forcément grave.

En plus, à un autre niveau, une telle structure pose des problèmes considérables de maintenance : car en cas de modification du code, il va falloir traquer toutes les apparitions plus ou moins identiques de ce code pour faire convenablement la modification ! Et si l'on en oublie une, patatras, on a laissé un bug.

Il faut donc 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 (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).

Reprenons un exemple de question à laquelle l'utilisateur doit répondre par oui ou par non.

Mauvaise Structure :

```
...
Ecrire "Etes-vous marié ?"
Rep1 ← ""
TantQue Rep1 <> "Oui" et Rep1 <> "Non"
  Ecrire "Tapez Oui ou Non"
  Lire Rep1
FinTantQue
...
Ecrire "Avez-vous des enfants ?"
Rep2 ← ""
TantQue Rep2 <> "Oui" et Rep2 <> "Non"
  Ecrire "Tapez Oui ou Non"
  Lire Rep2
FinTantQue
...
```

On le voit bien, il y a là une répétition quasi identique du traitement à accomplir. A chaque fois, on demande une réponse par Oui ou Non, avec contrôle de saisie. La seule chose qui change, c'est le nom de la variable dans laquelle on range la réponse. Alors, il doit bien y avoir un truc.

La solution, on vient de le voir, consiste à **isoler les instructions** demandant une réponse par Oui ou Non, et à appeler ces instructions à chaque fois que nécessaire. Ainsi, on évite les répétitions inutiles, et on a découpé notre problème en petits morceaux autonomes.

Nous allons donc créer une **fonction** dont le rôle sera de **renvoyer** la réponse (oui ou non) de l'utilisateur. Ce mot de "fonction", en l'occurrence, ne doit pas nous surprendre : nous avons étudié précédemment des fonctions fournies avec le langage, et nous avons vu que le but d'une fonction était de renvoyer une valeur. Eh bien, c'est exactement la même chose ici, sauf que c'est nous qui allons créer notre propre fonction, que nous appellerons RepOuiNon :

```
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. Cette valeur renvoyée par la fonction (ici, la valeur de la variable Truc) est en quelque sorte contenue dans le nom de la fonction lui-même, exactement comme c'était le cas dans les fonctions prédéfinies.

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 :

```
Bonne structure :

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

Et le tour est joué ! On a ainsi évité les répétitions inutiles, et si d'aventure, il y avait un bug dans notre contrôle de saisie, il suffirait de faire une seule correction dans la fonction RepOuiNon pour que ce bug soit éliminé de toute l'application. Elle n'est pas belle, la vie ?

Toutefois, les plus sagaces d'entre vous auront remarqué, tant dans le titre de la fonction que dans chacun des appels, la présence de **parenthèses**. Celles-ci, dès qu'on déclare ou qu'on appelle une fonction, sont obligatoires. Et si vous avez bien compris tout ce qui précède, vous devez avoir une petite idée de ce qu'on va pouvoir mettre dedans...

1.2 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 langage algorithmique, on dira que **le message devient un argument (ou un paramètre) de la fonction**. Cela n'est certes pas une découverte pour vous : nous avons longuement utilisé les arguments à propos des fonctions prédéfinies. Eh bien, quitte à construire nos propres fonctions, nous pouvons donc construire nos propres arguments. Voilà comment l'affaire se présente...

La fonction sera 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 ?")
...
```

Et voilà le travail.

Une remarque importante : là, on n'a passé qu'un seul argument en entrée. Mais bien entendu, on peut en passer autant qu'on veut, et créer des fonctions avec deux, trois, quatre, etc. arguments ; Simplement, il faut éviter d'être gourmands, et il suffit de passer ce dont on en a besoin, ni plus, ni moins !

Dans le cas que l'on vient de voir, le passage d'un argument à la fonction était élégant, mais pas indispensable. La preuve, cela marchait déjà très bien avec la première version. Mais on peut imaginer des situations où il faut absolument concevoir la fonction de sorte qu'on doive lui transmettre un certain nombre d'arguments si l'on veut qu'elle puisse remplir sa tâche. Prenons, par exemple, toutes les fonctions qui vont effectuer des calculs. Que ceux-ci soient simples ou compliqués, il va bien falloir envoyer à la fonction les valeurs grâce auxquelles elle sera censé produire son résultat (pensez tout bêtement à une fonction sur le modèle d'Excel, telle que celle qui doit calculer une somme ou une moyenne). C'est également vrai des fonctions qui traiteront des chaînes de caractères. Bref, dans 99% des cas, lorsqu'on créera une fonction, celle-ci devra comporter des arguments.

1.3 Deux mots sur l'analyse fonctionnelle

Comme souvent en algorithmique, si l'on s'en tient à la manière dont marche l'outil, tout cela n'est en réalité pas très compliqué. Les fonctions personnalisées se déduisent très logiquement de la manière nous nous avons déjà expérimenté les fonctions prédéfinies.

Le plus difficile, mais aussi le plus important, c'est d'acquérir le réflexe de **constituer systématiquement les fonctions adéquates quand on doit traiter un problème donné**, et de flairer la bonne manière de découper son algorithme en différentes fonctions pour le rendre léger, lisible et performant.

Cette partie de la réflexion s'appelle d'ailleurs l'**analyse fonctionnelle** d'un problème, et c'est toujours par elle qu'il faut commencer : en gros, dans un premier temps, on découpe le traitement en modules (algorithmique fonctionnelle), et dans un deuxième temps, on écrit chaque module (algorithmique classique). Cependant, avant

d'en venir là, il nous faut découvrir deux autres outils, qui prennent le relais là où les fonctions deviennent incapables de nous aider.

2. SOUS-PROCEDURES

2.1 Généralités

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. Vous ne voyez pas de quoi je veux parler ? Prenons deux exemples.

Premier exemple. Imaginons qu'au cours de mon application, j'aie plusieurs fois besoin d'effacer l'écran et de réafficher un bidule comme un petit logo en haut à gauche. On pourrait se dire qu'il faut créer une fonction pour faire cela. Mais quelle serait la valeur renvoyée par la fonction ? Aucune ! Effacer l'écran, ce n'est pas produire un résultat stockable dans une variable, et afficher un logo non plus. Voilà donc une situation ou j'ai besoin de répéter du code, mais où ce code n'a pas comme rôle de produire une valeur.

Deuxième exemple. Au cours de mon application, je dois plusieurs fois faire saisir un tableau d'entiers (mais à chaque fois, un tableau différent). Là encore, on serait tenté d'effectuer toutes ces saisies de tableaux dans une seule fonction. Mais problème, une fonction ne peut renvoyer qu'une seule valeur à la fois. Elle ne peut donc renvoyer un tableau, qui est une série de valeurs distinctes.

Alors, dans ces deux cas, faute de pouvoir traiter l'affaire par une fonction, devra-t-on en rester au code répétitif dont nous venons de dénoncer si vigoureusement les faiblesses ? Mmmmmh ? Vous vous doutez bien que non. Heureusement, tout est prévu, il y a une solution. Et celle-ci consiste à utiliser des **sous-procédures**.

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( ... )
...
Fin Procédure
```

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

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

Établissons un premier état des lieux.

- Alors qu'une fonction se caractérisait par les mots-clés **Fonction ... Fin Fonction**, une sous-procédure est identifiée par les mots-clés **Procédure ... Fin Procédure**. Oui, je sais, c'est un peu trivial comme remarque, mais, bon, on ne sait jamais.
- Lorsqu'une fonction était appelée, sa valeur (retournée) était toujours affectée à une variable (ou intégrée dans le calcul d'une expression). L'appel à une procédure, lui, est au contraire toujours une **instruction autonome**. "Exécute la procédure Bidule" est un ordre qui se suffit à lui-même.
- Toute fonction devait, pour cette raison, comporter l'instruction "Renvoyer". Pour la même raison, **l'instruction "Renvoyer" n'est jamais utilisée dans une sous-procédure**. La fonction est une valeur calculée, qui renvoie son résultat vers la procédure principale. La sous-procédure, elle, est un traitement ; elle ne "vaut" rien.
- Même une fois qu'on a bien compris les trois premiers points, on n'est pas complètement au bout de nos peines.

2.2 Le problème des arguments

En effet, il nous reste à examiner ce qui peut bien se trouver dans les parenthèses, à la place des points de suspension, aussi bien dans la déclaration de la sous-procédure que dans l'appel. Vous vous en doutez bien : c'est là que vont se trouver les outils qui vont permettre l'échange d'informations entre la procédure principale et la sous-procédure (en fait, cette dernière phrase est trop restrictive : mieux vaudrait dire : entre la procédure appelante et la procédure appelée. Car une sous-procédure peut très bien en appeler elle-même une autre afin de pouvoir accomplir sa tâche)

De même qu'avec les fonctions, les valeurs qui circulent depuis la procédure (ou la fonction) appelante vers la sous-procédure appelée se nomment des **arguments**, ou des **paramètres en entrée** de la sous-procédure. Comme on le voit, qu'il s'agisse des sous-procédure ou des fonctions, ces choses jouant exactement le même rôle (transmettre une information depuis le code donneur d'ordres jusqu'au code sous-traitant), elle portent également le même nom. Unique petite différence, on a précisé cette fois qu'il s'agissait d'arguments, ou de paramètres, **en entrée**. Pourquoi donc ?

Tout simplement parce que que dans une sous-procédure, on peut être amené à vouloir renvoyer des résultats vers le programme principal ; or, là, à la différence des fonctions, rien n'est prévu : la sous-procédure, en tant que telle, ne "renvoie" rien du tout (comme on vient de le voir, elle est d'ailleurs dépourvue de l'instruction "renvoyer"). Ces résultats que la sous-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 en sortie**.

Ceci nous permet de reformuler en d'autres termes la vérité fondamentale apprise un peu plus haut : **toute sous-procédure possédant un et un seul paramètre en sortie peut également être écrite sous forme d'une fonction** (et entre nous, c'est une formulation préférable car un peu plus facile à comprendre et donc à retenir).

Jusque là, ça va ? Si oui, prenez un cachet d'aspirine et poursuivez la lecture. Si non, prenez un cachet d'aspirine et recommencez depuis le début. Et dans les deux cas, n'oubliez pas le grand verre d'eau pour faire passer l'aspirine.

Il nous reste un détail à examiner, détail qui comme vous vous en doutez bien, a une certaine importance : comment fait-on pour faire comprendre à un langage quels sont les paramètres qui doivent fonctionner en entrée et quels sont ceux qui doivent fonctionner en sortie...

2.3 Comment ça marche tout ça ?

En fait, si je dis qu'un paramètre est "en entrée" ou "en sortie", j'énonce quelque chose à propos de son rôle dans le programme. Je dis ce que je souhaite qu'il fasse, la manière dont je veux qu'il se comporte. Mais les programmes eux-mêmes n'ont cure de mes désirs, et ce n'est pas cette classification qu'ils adoptent. C'est toute la différence entre dire qu'une prise électrique sert à brancher un rasoir ou une cafetière (ce qui caractérise son rôle), et dire qu'elle est en 220 V ou en 110 V (ce qui caractérise son type technique, et qui est l'information qui intéresse l'électricien). A l'image des électriciens, les langages se contrefichent de savoir quel sera le rôle (entrée ou sortie) d'un paramètre. Ce qu'ils exigent, c'est de connaître leur voltage... pardon, le **mode de passage** de ces paramètres. Il n'en existe que deux :

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

...Voyons de plus près de quoi il s'agit.

Reprenons l'exemple que nous avons déjà utilisé plus haut, celui de notre fonction RepOuiNon. Comme nous l'avons vu, rien ne nous empêche de réécrire cette fonction sous la forme d'une procédure (puisqu'une fonction n'est qu'un cas particulier de sous-procédure). Nous laisserons pour l'"instant de côté la question de savoir comment

renvoyer la réponse (contenue dans la variable Truc) vers le programme principal. En revanche, nous allons déclarer que Msg est un paramètre dont la transmission doit se faire par valeur. Cela donnera la chose suivante :

```
Procédure RepOuiNon(Msg en Caractère par valeur)
Ecrire Msg
Truc ← ""
TantQue Truc <> "Oui" et Truc <> "Non"
  Ecrire "Tapez Oui ou Non"
  Lire Truc
FinTantQue
??? Comment transmettre Truc à la procédure appelante ???
Fin Procédure
```

Quant à l'appel à cette sous-procédure, il pourra prendre par exemple cette forme :

```
M ← "Etes-vous marié ?"
Appeler RepOuiNon(M)
```

Que va-t-il se passer ?

Lorsque le programme principal arrive sur la première ligne, il affecte la variable M avec le libellé "Êtes-vous marié". La ligne suivante déclenche l'exécution de la sous-procédure. Celle-ci crée aussitôt une variable Msg. Celle-ci ayant été déclarée comme un paramètre passé **par valeur**, Msg va être affecté avec le même contenu que M. Cela signifie que **Msg est dorénavant une copie de M**. Les informations qui étaient contenues dans M ont été intégralement recopiées (en double) dans Msg. Cette copie subsistera tout au long de l'exécution de la sous-procédure RepOuiNon et sera détruite à la fin de celle-ci.

Une conséquence essentielle de tout cela est que si d'aventure la sous-procédure RepOuiNon contenait une instruction qui modifiait le contenu de la variable Msg, cela n'aurait aucune espèce de répercussion sur la procédure principale en général, et sur la variable M en particulier. **La sous-procédure ne travaillant que sur une copie de la variable qui a été fournie par le programme principal, elle est incapable, même si on le souhaitait, de modifier la valeur de celle-ci**. Dit d'une autre manière, dans une procédure, **un paramètre passé par valeur ne peut être qu'un paramètre en entrée**.

C'est en même temps une limite (aggravée par le fait que les informations ainsi recopiées occupent dorénavant deux fois plus de place en mémoire) et une sécurité : quand on transmet un paramètre par valeur, on est sûr et certain que même en cas de bug dans la sous-procédure, la valeur de la variable transmise ne sera jamais modifiée par erreur (c'est-à-dire écrasée) dans le programme principal.

Admettons à présent que nous déclarions un second paramètre, Truc, en précisant cette fois qu'il sera transmis **par référence**. Et adoptons pour la procédure l'écriture suivante :

```
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
```

L'appel à la sous-procédure deviendrait par exemple :

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

Dépiautons le mécanisme de cette nouvelle écriture. En ce qui concerne la première ligne, celle qui affecte la variable M, rien de nouveau sous le soleil. Toutefois, l'appel à la sous-procédure provoque deux effets très différents. Comme on l'a déjà dit, la variable Msg est créée et immédiatement affectée avec une copie du contenu de M, puisqu'on a exigé un passage par valeur. Mais en ce qui concerne Truc, il en va tout autrement. Le fait qu'il s'agisse cette fois d'un passage par référence fait que **la variable Truc ne contiendra pas la valeur de T, mais son adresse, c'est-à-dire sa référence**.

Dès lors, **toute modification de Truc sera immédiatement redirigée, par ricochet en quelque sorte, sur T**. Truc n'est pas une variable ordinaire : elle ne contient pas de valeur, mais seulement la référence à une valeur, qui elle, se trouve ailleurs (dans la variable T). Il s'agit donc d'un genre de variable complètement nouveau, et différent de ce que nous avons vu jusque là. Ce type de variable porte un nom : on l'appelle **un pointeur**. Tous les paramètres passés par référence sont des pointeurs, mais les pointeurs ne se limitent pas aux paramètres passés par référence (même si ce sont les seuls que nous verrons dans le cadre de ce cours). Il faut bien comprendre que ce type de variable étrange est géré directement par les langages : **à partir du moment où une variable est considérée comme un pointeur, toute affectation de cette variable se traduit automatiquement par la modification de la variable sur laquelle elle pointe**.

Passer un paramètre par référence, cela présente donc deux avantages. Et d'une, 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 de deux, **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

Mais alors, demanderez-vous dans un élan de touchante naïveté, si le passage par référence présente les deux avantages présentés il y a un instant, pourquoi ne pas s'en servir systématiquement ? Pourquoi s'embêter avec les passages par valeur, qui non seulement utilisent de la place en mémoire, mais qui de surcroît nous interdisent d'utiliser la variable comme un paramètre en sortie ?

Eh bien, justement, parce qu'on ne pourra pas utiliser comme paramètre en sortie, et que cet inconvénient se révèle être aussi, éventuellement, un avantage. Disons la chose autrement : **c'est une sécurité**. C'est la garantie que quel que soit le bug qui pourra affecter la sous-procédure, ce bug ne viendra jamais mettre le foutoir dans les variables du programme principal qu'elle ne doit pas toucher. Voilà pourquoi, lorsqu'on souhaite définir un paramètre dont on sait qu'il fonctionnera exclusivement en entrée, il est sage de le verrouiller, en quelque sorte, en le définissant comme passé par valeur. Et Lycée de Versailles, ne seront définis comme passés par référence que les paramètres dont on a absolument besoin qu'ils soient utilisés en sortie.

3. VARIABLES PUBLIQUES ET PRIVEES

Résumons la situation. Nous venons de voir que nous pouvions découper un long traitement comportant éventuellement des redondances (notre application) en différents modules. Et nous avons vu que les informations pouvaient être transmises entre ces modules selon deux modes :

- si le module appelé est une fonction, par le **retour du résultat**
- dans tous les cas, par la **transmission de paramètres** (que ces paramètres soient passés par valeur ou par référence)

En fait, il existe un troisième et dernier 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. Si je crée une variable Toto dans une procédure Bidule, et qu'en cours de route, ma

procédure Bidule appelle une sous-procédure Machin, il est hors de question que Machin puisse accéder à Toto, ne serait-ce que pour connaître sa valeur (et ne parlons pas de la modifier). 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 Truc, 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 faites est évidemment fonction de chaque langage de programmation. En pseudo-code algorithmique, on pourra utiliser le mot-clé **Publique** :

```
Variable Publique Toto en Numérique
```

Alors, pourquoi ne pas rendre toutes les variables publiques, et s'épargner ainsi de fastidieux efforts pour passer des paramètres ? C'est très simple, et c'est toujours la même chose : **les variables globales consomment énormément de ressources en mémoire**. En conséquence, le principe qui doit présider au choix entre variables publiques et privées doit être celui de **l'économie de moyens** : on ne déclare comme publiques que les variables qui doivent absolument l'être. Et chaque fois que possible, lorsqu'on crée une sous-procédure, on utilise le passage de paramètres plutôt que des variables publiques.

4. PEUT-ON TOUT FAIRE ?

A cette question, la réponse est bien évidemment : oui, on peut tout faire. Mais c'est précisément la raison pour laquelle on peut vite en arriver à faire aussi absolument n'importe quoi.

N'importe quoi, c'est quoi ? C'est par exemple, comme on vient de le voir, mettre des variables globales partout, sous prétexte que c'est autant de paramètres qu'on n'aura pas à passer.

Mais on peut imaginer d'autres atrocités.

Par exemple, une fonction, dont un des paramètres d'entrée serait passé par référence, et modifié par la fonction. Ce qui signifierait que cette fonction produirait non pas un, mais deux résultats. Autrement dit, que sous des dehors de fonctions, elle se comporterait en réalité comme une sous-procédure.

Ou inversement, on peut concevoir une procédure qui modifierait la valeur d'un paramètre (et d'un seul) passé par référence. Il s'agirait là d'une procédure qui en réalité, serait une fonction. Quoique ce dernier exemple ne soit pas d'une gravité dramatique, il participe de la même logique consistant à embrouiller le code en faisant passer un outil pour un autre, au lieu d'adopter la structure la plus claire et la plus lisible possible.

Enfin, il ne faut pas écarter la possibilité de programmeurs particulièrement vicieux, qui par un savant mélange de paramètres passés par référence, de variables globales, de procédures et de fonctions mal choisies, finiraient par accoucher d'un code absolument illogique, illisible, et dans lequel la chasse à l'erreur relèverait de l'exploit.

Trèfle de plaisanteries : le principe qui doit guider tout programmeur est celui de la solidité et de la clarté du code. **Une application bien programmée est une application à l'architecture claire, dont les différents modules font ce qu'ils disent, disent ce qu'il 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**. Celles-ci doivent être employées avec nos célèbres amis italo-arméniens, c'est-à-dire avec parcimonie et à bon escient.
2. de **regrouper sous forme de modules distincts** tous les morceaux de code qui possèdent une certaine unité fonctionnelle (programmation par "blocs"). C'est-à-dire de faire la chasse aux lignes de codes redondantes, ou quasi-redondantes.
3. de faire de ces modules **des fonctions lorsqu'ils renvoient un résultat unique**, et **des sous-procédures dans tous les autres cas** (ce qui implique de ne **jamais** passer un paramètre par référence à une fonction : soit on n'en a pas besoin, soit on en a besoin, et ce n'est alors plus une fonction).

Respecter ces règles d'hygiène est indispensable si l'on veut qu'une application ressemble à autre chose qu'au palais du facteur Cheval. Car une architecture à laquelle on ne comprend rien, c'est sans doute très poétique, mais il y a des circonstances où l'efficacité est préférable à la poésie. Et, pour ceux qui en douteraient encore, la programmation informatique fait (hélas ?) partie de ces circonstances.

5. ALGORITHMES FONCTIONNELS

Pour clore ce chapitre, voici quelques mots supplémentaires à propos de la structure générale d'une application. Comme on l'a dit à plusieurs reprises, celle-ci va couramment être formée d'une procédure principale, et de fonctions et de sous-procédures (qui vont au besoin elles-mêmes en appeler d'autres, etc.). L'exemple typique est celui d'un menu, ou d'un sommaire, qui « branche » sur différents traitements, donc différentes sous-procédures.

L'algorithme fonctionnel de l'application est le **découpage** et/ou la **représentation graphique** de cette structure générale, ayant comme objectif de faire comprendre d'un seul coup d'œil quelle procédure fait quoi, et quelle procédure appelle quelle autre. L'algorithme fonctionnel est donc en quelque sorte la construction du squelette de l'application. Il se situe à un niveau plus général, plus abstrait, que l'algorithme normal, qui lui, détaille pas à pas les traitements effectués au sein de chaque procédure.

Dans la construction – et la compréhension – d'une application, les deux documents sont indispensables, et constituent deux étapes successives de l'élaboration d'un projet. La troisième – et dernière – étape, consiste à écrire, pour chaque procédure et fonction, l'algorithme détaillé.

Exemple de réalisation d'un algorithme fonctionnel : Le Jeu du Pendu

Vous connaissez tous ce jeu : l'utilisateur doit deviner un mot choisi au hasard par l'ordinateur, en un minimum d'essais. Pour cela, il propose des lettres de l'alphabet. Si la lettre figure dans le mot à trouver, elle s'affiche. Si elle n'y figure pas, le nombre des mauvaises réponses augmente de 1. Au bout de dix mauvaises réponses, la partie est perdue.

Ce petit jeu va nous permettre de mettre en relief les trois étapes de la réalisation d'un algorithme un peu complexe ; bien entendu, on pourrait toujours ignorer ces trois étapes, et se lancer comme un dératé directement dans la gueule du loup, à savoir l'écriture de l'algorithme définitif. Mais, sauf à être particulièrement doué, mieux vaut respecter le canevas qui suit, car les difficultés se résolvent mieux quand on les saucissonne…

Etape 1 : le dictionnaire des données

Le but de cette étape est d'identifier les informations qui seront nécessaires au traitement du problème, et de choisir le type de codage qui sera le plus satisfaisant pour traiter ces informations. C'est un moment essentiel de la réflexion, qu'il ne faut surtout pas prendre à la légère… Or, neuf programmeurs débutants sur dix bâclent cette réflexion, quand ils ne la zappent pas purement et simplement. La punition ne se fait généralement pas attendre longtemps ; l'algorithme étant bâti sur de mauvaises fondations, le programmeur se rend compte tout en l'écrivant que le choix de codage des informations, par exemple, mène à des impasses. La précipitation est donc punie par le fait qu'on est obligé de tout reprendre depuis le début, et qu'on a au total perdu bien davantage de temps qu'on en a cru en gagner…

Donc, avant même d'écrire quoi que ce soit, les questions qu'il faut se poser sont les suivantes :

- de quelles informations le programme va-t-il avoir besoin pour venir à bout de sa tâche ?
- pour chacune de ces informations, quel est le meilleur codage ? Autrement dit, celui qui sans gaspiller de la place mémoire, permettra d'écrire l'algorithme le plus simple ?

Encore une fois, il ne faut pas hésiter à passer du temps sur ces questions, car certaines erreurs, ou certains oublis, se payent cher par la suite. Et inversement, le temps investi à ce niveau est largement rattrapé au moment du développement proprement dit.

Pour le jeu du pendu, voici la liste des informations dont on va avoir besoin :

- une liste de mots (si l'on veut éviter que le programme ne propose toujours le même mot à trouver, ce qui risquerait de devenir assez rapidement lassant…)
- le mot à deviner
- la lettre proposée par le joueur à chaque tour
- le nombre actuel de mauvaises réponses
- et enfin, last but not least, l'ensemble des lettres déjà trouvées par le joueur. Cette information est capitale ; le programme en aura besoin au moins pour deux choses : d'une part, pour savoir si le mot entier a été trouvé. D'autre part, pour afficher à chaque tour l'état actuel du mot (je rappelle qu'à chaque tour, les lettres trouvées sont affichées en clair par la machine, les lettres restant à deviner étant remplacées par des tirets).
- à cela, on pourrait ajouter une liste comprenant l'ensemble des lettres déjà proposées par le joueur, qu'elles soient correctes ou non ; ceci permettra d'interdire au joueur de proposer à nouveau une lettre précédemment jouée.

Cette liste d'informations n'est peut-être pas exhaustive ; nous aurons vraisemblablement besoin au cours de l'algorithme de quelques variables supplémentaires (des compteurs de boucles, des variables temporaires, etc.). Mais les informations essentielles sont bel et bien là. Se pose maintenant le problème de choisir le mode de codage le plus futé. Si, pour certaines informations, la question va être vite réglée, pour d'autres, il va falloir faire des choix (et si possible, des choix intelligents !). C'est parti, mon kiki :

- Pour la liste des mots à trouver, il s'agit d'un ensemble d'informations de type alphanumérique. Ces informations pourraient faire partie du corps de la procédure principale, et être ainsi stockées en mémoire vive, sous la forme d'un tableau de chaînes. Mais ce n'est certainement pas le plus judicieux. Toute cette place occupée risque de peser lourd inutilement, car il n'y a aucun intérêt à stocker l'ensemble des mots en mémoire vive. Et si l'on souhaite enrichir la liste des mots à trouver, on sera obligé de réécrire des lignes de programme… Conclusion, la liste des mots sera bien plus à sa place dans un fichier texte, dans lequel le programme ira piocher un seul mot, celui qu'il faudra trouver. Nous constituerons donc un fichier texte, appelé dico.txt, dans lequel figurera un mot par ligne (par enregistrement).
- Le mot à trouver, lui, ne pose aucun problème : il s'agit d'une information simple de type chaîne, qui pourra être stocké dans une variable appelée mot, de type caractère.
- De même, la lettre proposée par le joueur est une information simple de type chaîne, qui sera stockée dans une variable appelée lettre, de type caractère.
- Le nombre actuel de mauvaises réponses est une information qui pourra être stockée dans une variable numérique de type entier simple appelée MovRep.
- L'ensemble des lettres trouvées par le joueur est typiquement une information qui peut faire l'objet de plusieurs choix de codage ; rappelons qu'au moment de l'affichage, nous aurons besoin de savoir pour chaque lettre du mot à deviner si elle a été trouvée ou non. Une première possibilité, immédiate, serait de disposer d'une chaîne de caractères comprenant l'ensemble des lettres précédemment trouvées. Cette solution est loin d'être mauvaise, et on pourrait tout à fait l'adopter. Mais ici, on fera une autre choix, ne serait-ce que pour varier les plaisirs : on va se doter d'un tableau de booléens, comptant autant d'emplacements qu'il y a de lettres dans le mot à deviner. Chaque emplacement du tableau correspondra à une lettre du mot à trouver, et indiquera par sa valeur si la lettre a été découverte ou non (faux, la lettre n'a pas été devinée, vrai, elle l'a été). La correspondance entre les éléments du tableau et le mot à deviner étant immédiate, la programmation de nos boucles en sera facilitée. Nous baptiserons notre tableau de booléens du joli nom de « verif ».

- Enfin, l'ensemble des lettres proposées sera stockée sans soucis dans une chaîne de caractères nommée Propos.

Nous avons maintenant suffisamment gambergé pour dresser le tableau final de cette étape, à savoir le dictionnaire des données proprement dit :

.

Nom	Type	Description
Dico.txt	Fichier texte	Liste des mots à deviner
Mot	Caractère	Mot à deviner
Lettre	Caractère	Lettre proposée
MovRep	Entier	Nombre de mauvaises réponses
Verif()	Tableau de Booléens	Lettres précédemment devinées, en correspondance avec Mot
Propos	Caractère	Liste des lettres proposées

Etape 2 : l'algorithme fonctionnel

On peut à présent passer à la réalisation de l'algorithme fonctionnel, c'est-à-dire au découpage de notre problème en blocs logiques. Le but de la manœuvre est multiple :

- faciliter la réalisation de l'algorithme définitif en le tronçonnant en plus petits morceaux.
- Gagner du temps et de la légèreté en isolant au mieux les sous-procédures et fonctions qui méritent de l'être. Eviter ainsi éventuellement des répétitions multiples de code au cours du programme, répétitions qui ne diffèrent les unes des autres qu'à quelques variantes près.
- Permettre une division du travail entre programmeurs, chacun se voyant assigner la programmation de sous-procédures ou de fonctions spécifiques (cet aspect est essentiel dès qu'on quitte le bricolage personnel pour entrer dans le monde de la programmation professionnelle, donc collective).

Dans notre cas précis, un premier bloc se détache : il s'agit de ce qu'on pourrait appeler les préparatifs du jeu (choix du mot à deviner). Puisque le but est de renvoyer une valeur et une seule (le mot choisi par la machine), nous pouvons confier cette tâche à une fonction spécialisée ChoixDuMot (à noter que ce découpage est un choix de lisibilité, et pas une nécessité absolue ; on pourrait tout aussi bien faire cela dans la procédure principale).

Cette procédure principale, justement, va ensuite avoir nécessairement la forme d'une boucle Tantque : en effet , tant que la partie n'est pas finie, on recommence la série des traitements qui représentent un tour de jeu. Mais comment, justement, savoir si la partie est finie ? Elle peut se terminer soit parce que le nombre de mauvaises réponses a atteint 10, soit parce que toutes les lettres du mot ont été trouvées. Le mieux sera donc de confier l'examen de tout cela à une fonction spécialisée, PartieFinie, qui renverra une valeur numérique (0 pour signifier que la partie est en cours, 1 en cas de victoire, 2 en cas de défaite).

Passons maintenant au tour de jeu.

La première chose à faire, c'est d'afficher à l'écran l'état actuel du mot à deviner : un mélange de lettres en clair (celles qui ont été trouvées) et de tirets (correspondant aux lettres non encore trouvées). Tout ceci pourra être pris en charge par une sous-procédure spécialisée, appelée AffichageMot. Quant à l'initialisation des différentes variables, elle pourra être placée, de manière classique, dans la procédure principale elle-même.

Ensuite, on doit procéder à la saisie de la lettre proposée, en veillant à effectuer les contrôles de saisie adéquats. Là encore, une fonction spécialisée, SaisieLettre, sera toute indiquée.

Une fois la proposition faite, il convient de vérifier si elle correspond ou non à une lettre à deviner, et à en tirer les conséquences. Ceci sera fait par une sous-procédure appelée VérifLettre.

Enfin, une fois la partie terminée, on doit afficher les conclusions à l'écran ; on déclare à cet effet une dernière procédure, FinDePartie.

Nous pouvons, dans un algorithme fonctionnel complet, dresser un tableau des différentes procédures et fonctions, exactement comme nous l'avons fait juste avant pour les données (on s'épargnera cette peine dans le cas présent, ce que nous avons écrit ci-dessus suffisant amplement. Mais dans le cas d'une grosse application, un tel travail serait nécessaire et nous épargnerait bien des soucis).

On peut aussi schématiser le fonctionnement de notre application sous forme de blocs, chacun des blocs représentant une fonction ou une sous-procédure :

A ce stade, l'analyse dite fonctionnelle est terminée. Les fondations (solides, espérons-le) sont posées pour finaliser l'application.

Etape 3 : Algorithmes détaillés

Normalement, il ne nous reste plus qu'à traiter chaque procédure isolément. On commencera par les sous-procédures et fonctions, pour terminer par la rédaction de la procédure principale.

ATTENTION ! les liens ci-dessous mènent directement aux corrigés !

Fonction ChoixDuMot

Fonction PartieFinie

Procédure AffichageMot

Procédure SaisieLettre

Procédure VérifLettre

Procédure Epilogue

Procédure Principale

PARTIE 12
NOTIONS COMPLEMENTAIRES

Une fois n'est pas coutume, ce chapitre ne sera l'objet d'aucun exercice. Cela ne veut pas dire pour autant que ce qui s'y trouve n'est pas intéressant.

Non mais des fois.

1. PROGRAMMATION STRUCTUREE

Petit retour sur une notion très rapidement survolée plus haut : celle de « programmation structurée ». En fait, nous avons jusqu'à présent, tels Monsieur Jourdain, fait de la programmation structurée sans le savoir. Aussi, plutôt qu'expliquer longuement en quoi cela consiste, je préfère prendre le problème par l'autre bout : en quoi cela ne consiste pas.

Dans certains langages (historiquement, ce sont souvent des langages anciens), les lignes de programmation portent des numéros. Et les lignes sont exécutées par la machine dans l'ordre de ces numéros. Jusqu'ici, en soi, pas de problème. Mais l'astuce est que tous ces langages, il existe une instruction de branchement, notée **aller à** en pseudo-code, instruction qui envoie directement le programme à la ligne spécifiée. Inversement, ce type de langage ne comporte pas d'instructions comme FinTantQue, ou FinSi, qui « ferment » un bloc.

Prenons l'exemple d'une structure « Si … Alors … Sinon »

```
Programmation Structurée
Si condition Alors
  instructions 1
Sinon
  instructions 2
FinSi

Programmation non structurée
1000 Si condition Alors Aller En 1200
1100 instruction 1
1110 etc.
1120 etc.
1190 Aller en 1400
1200 instruction 2
1210 etc.
1220 etc.
1400 suite de l'algorithme
```

Vous voyez le topo : un programme écrit dans ce type de langages se présente comme **une suite de branchements emmêlés les uns dans les autres**. D'une part, on ne peut pas dire que cela favorise la lisibilité du programme. D'autre part, c'est une source importante d'erreurs, car tôt ou tard on oublie un « aller à », ou on un met un de trop, etc. A fortiori lorsqu'on complique un algorithme existant, cela peut devenir un jungle inextricable.

A l'inverse, la programmation structurée, surtout si l'on prend soin de rationaliser la présentation en mettant des lignes de commentaires et en pratiquant l'indentation, évite des erreurs, et révèle sa structure logique de manière très claire.

Le danger est que si la plupart des langages de programmation utilisés sont structurés, ils offrent tout de même la plupart du temps la possibilité de pratiquer la programmation non structurée. Dans ce cas, les lignes ne sont pas désignées par des numéros, mais certaines peuvent être repérées par des noms (dits « étiquettes ») et on dispose d'une instruction de branchement.

Une règle d'hygiène absolue est de programmer systématiquement de manière structurée, sauf impératif contraire fixé par le langage (ce qui est aujourd'hui de plus en plus rare).

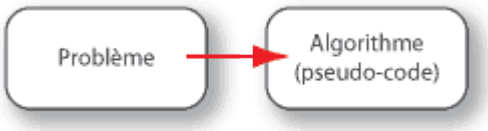
Autrement dit, même quand un langage vous offre une possibilité de faire des entorses à la programmation structurée, il ne faut s'en saisir sous aucun prétexte.

2. INTERPRETATION ET COMPILATION

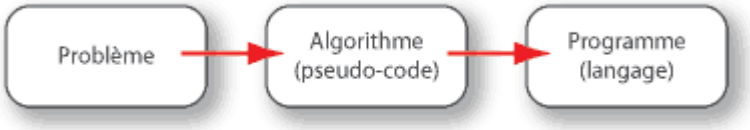
Avec ce paragraphe, on sort un peu de l'algorithmique proprement dite pour entrer dans le domaine plus technique de la réalisation pratique. Ou, si l'on préfère, ces dernières lignes sont l'apothéose, le bouquet final, l'extase ultime, la consécration grandiose, de ce cours.

En toute modestie, bien sûr.

Jusqu'ici, nous avons travaillé sur la première étape de la réalisation d'un programme : la rédaction de l'algorithme.



En fait, 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 (à la rigueur en hexadécimal) et qui s'appelle le langage machine (ou assembleur).

C'est à cela que sert un langage : à vous épargner la programmation en binaire (une pure horreur, vous vous en doutez) et vous permettre de vous faire comprendre de l'ordinateur d'une manière (relativement) lisible.

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 : le gain est couramment d'un facteur 10, voire 20 ou plus.

Toute application destinée à un usage professionnel (ou même, tout simplement sérieux) est forcément une application compilée.

3. UNE LOGIQUE VICELARDE : LA PROGRAMMATION RECURSIVE

Vous savez comment sont les informaticiens : on ne peut pas leur donner quoi que ce soit sans qu'ils essayent de jouer avec, et le pire, c'est qu'ils y réussissent.

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 vous expliquer de quoi il retourne, nous allons reprendre un exemple cher à vos cœurs : le calcul d'une factorielle (là, je sentais que j'allais encore me faire des copains).

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 avions programmé cela aussi sec avec une boucle Pour, et roule Raoul. Mais une autre manière de voir les choses, ni plus juste, ni moins juste, serait de dire que quel que soit le nombre n :

n ! = n x (n-1) !

En bon français : la factorielle d'un nombre, c'est ce nombre multiplié par la factorielle du nombre précédent. Encore une fois, c'est une manière ni plus juste ni moins juste de présenter les choses ; c'est simplement une manière différente.

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.

Toutefois, il nous manque une chose pour finir : quand ces auto-appels de la fonction Fact vont-ils s'arrêter ? Cela n'aura-t-il donc jamais de fin ? Si, bien sûr, rassure-toi, ô public, la récursivité, ce n'est pas Les Feux de L'Amour. 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 en Numérique)
Si N = 0 alors
  Renvoyer 1
Sinon
  Renvoyer Fact(N-1) * N
Finsi
Fin Fonction
```

Vous remarquerez que le processus récursif remplace en quelque sorte la boucle, c'est-à-dire un processus itératif. Et en plus, avec tous ces nouveaux mots qui riment, vous allez pouvoir écrire de très chouettes poèmes. Vous remarquerez aussi 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.
- Last but not least, et c'est le gag final, **tout problème formulé en termes récursifs peut également être formulé en termes itératifs !** Donc, si la programmation récursive peut faciliter la vie du programmeur, elle n'est jamais indispensable. Mais ça me faisait tant plaisir de vous en parler que je n'ai pas pu résister… Et puis, accessoirement, même si on ne s'en sert pas, en tant qu'informaticien, il faut connaître cette technique sur laquelle on peut toujours tomber un jour ou l'autre.