REPUBLIQUE D'HAITI

Ministère de L'Éducation Nationale et de la Formation Professionnelle (MENFP)

Direction de l'Enseignement Secondaire (DES)

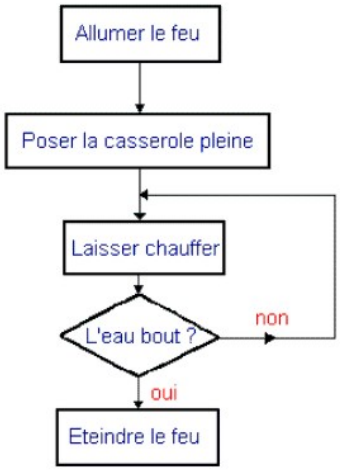


Fig. 5. - Organigramme pour faire bouillir de l'eau.

PROGRAMME DÉTAILLÉ ET MODULE

4 éme Année du Secondaire

Informatique

Octobre 2010

	THÈMES	COMPÉTENCES	CONTENUS	SUGGESTIONS D'ACTIVITES ENSEIGNEMENT/APPRENTISSAGE
Algorithme	***Les Variables Lecture et Ecriture***	✓ *A partir des exemples de situations problèmes réelles, faire comprendre la notion d'algorithmique en prenant le soin de détacher des problèmes les variables tout en tenant compte de leurs types et des expressions à utiliser.*	✓ *A quoi servent les variables* ✓ *Déclarations des variables* ✓ Les types de variables ✓ Notions d'affectations aux variables ✓ Expressions et Opérateurs ✓ *Les Instructions de Lecture et d'Ecriture*	*1.*
	Les Tests Encore de la Logique	✓ *Apprendre (à partir de situations problèmes) la logique de tests en utilisant les conditions de vie réelle.* ✓ *Savoir faire un test pour les différentes situations.*	✓ *Structure d'un Test* ✓ *Qu'est ce qu'une Condition ?* ✓ *Conditions Composés* ✓ *Test imbriqué* ✓ *De l'aiguille à la gare de tri* ✓ *Les Variables Booléennes* ✓ *Faut il mettre un ET ? Faut il mettre un OU ?*	*1.*
	Les Boucles	✓ *Savoir compter le nombre de tour à partir d'une boucle* ✓ *Utiliser un jeu de boucle pour faire des tests*	✓ *A quoi sert-il ?* ✓ Les boucles « Pour », « Tant que…Faire », « Faire …. Tant que » et le multi choix ✓ *Boucler en comptant, ou compter en bouclant* ✓ *Des boucles dans les boucles*	*1.*

	THÈMES	COMPÉTENCES	CONTENUS	SUGGESTIONS D'ACTIVITES ENSEIGNEMENT/APPRENTISSAGE
Programmation	***Introduction Les Variables***	✓ *Expliquer la notion de compilateur et apprendre les prémices du logiciel devc++* ✓ *Savoir exécuter un programme* ✓ *Déclarer les variables en c*	✓ *Les données et les résultats* ✓ *Le codage et le compilateur ?* ✓ *Comment Exécuter un programme* ✓ *Le nom et Les types de variables* ✓ *Instructions de déclaration d'une variable* ✓ *Valeur d'une Variable*	*1.*
	Communication entre le Programme et l'Utilisateur Instructions et Expressions	✓ *Maîtriser les méthodes d'affichage en c* ✓ *Savoir faire la saisie de données* ✓ *Connaître la syntaxe pour ne pas dépasser les limites du codage* ✓ *Maîtriser les conversions de types et les expressions*	✓ *Affichage à l'écran et Notions de précisions d'affichage* ✓ *Lecture au clavier* ✓ *Les opérateurs mathématiques* ✓ *Dépassement des limites du codage* ✓ *Incrémentation et décrémentation unitaire* ✓ *Particularité du type float* ✓ *Expressions simples et expressions mixtes* ✓ *Conversions forcées et le type char*	*1.*
	Structures Conditionnelles Structures Répétitives : les Boucles	✓ *Apprendre à utiliser les jeux de tests* ✓ *Utiliser les boucles pour en faire des tests*	✓ *Conditions composées et tests imbriqués* ✓ *Les boucles « While », « Do…While », « For » et le multi choix*	*1.*

MODULE

INTRODUCTION A L'ALGORITHMIQUE

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

Avez-vous déjà indiqué un chemin à un touriste égaré ? Avez-vous fait chercher un objet à quelqu'un par téléphone ? Avez-vous écrit 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 calculatrice ne veut pas fonctionner.

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

Le malheur (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 difficile à comprendre 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 être bon en maths pour être 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 à la place de la machine qui va les exécuter, pour 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 de chacun de vos algorithmes doit être considérée comme la moitié du travail à accomplir.

3. L'ADN et les ordinateurs

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 gérer 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 écrit-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, cela n'est plus pratique 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 1

LES VARIABLES

1. A quoi servent les variables ?

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

Pour employer une image, une variable est une **boîte**, que le programme (l'ordinateur) va repérer par une **étiquette**. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette.

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

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

2. Déclaration des variables

La première chose à faire avant de pouvoir utiliser une variable est de **créer la boîte et de lui coller une étiquette.** Ceci se fait tout au début de l'algorithme, avant même les instructions

proprement dites. C'est ce qu'on appelle la **déclaration des variables**. C'est un genre de déclaration certes moins romantique qu'une déclaration d'amour, mais d'un autre côté moins désagréable qu'une déclaration d'impôts.

Le **nom** de la variable (l'étiquette de la boîte) obéit à des impératifs changeant selon les langages. Toutefois, une règle absolue est qu'un nom de variable peut comporter des lettres et des chiffres, mais qu'il exclut la plupart des signes de ponctuation, en particulier les espaces. Un nom de variable correct commence également et impérativement par une lettre. Quant au nombre maximal de signes pour un nom de variable, il dépend du langage utilisé.

En pseudo-code algorithmique, on est bien sûr libre du nombre de signes pour un nom de variable, même si pour des raisons purement pratiques, on évite généralement les noms à rallonge.

Lorsqu'on déclare une variable, il ne suffit pas de créer une boîte (réserver un emplacement mémoire) ; encore doit-on préciser ce que l'on voudra mettre dedans, car de cela dépendent la **taille** de la boîte (de l'emplacement mémoire) et le **type de codage** utilisé.

2.1 Types numériques classiques

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

Si l'on réserve un octet pour coder un nombre, je rappelle qu'on ne pourra coder que $2^8 = 256$ valeurs différentes. Cela peut signifier par exemple les nombres entiers de 1 à 256, ou de 0 à 255, ou de –127 à +128… Si l'on réserve deux octets, on a droit à 65 536 valeurs ; avec trois octets, 16 777 216, etc. Et là se pose un autre problème : ce codage doit-il représenter des nombres décimaux ? Des nombres négatifs ?

Bref, le type de codage (autrement dit, le type de variable) choisi pour un nombre va déterminer :

- Les valeurs maximales et minimales des nombres pouvant être stockés dans la variable
- La précision de ces nombres (dans le cas de nombres décimaux).

Tous les langages, quels qu'ils soient offrent un « bouquet » de types numériques, dont le détail est susceptible de varier légèrement d'un langage à l'autre. Grosso modo, on retrouve cependant les types suivants :

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

Pourquoi ne pas déclarer toutes les variables numériques en réel double, histoire de bétonner et d'être certain qu'il n'y aura pas de problème ? En vertu du principe de **l'économie de moyens**. Un bon algorithme ne se contente pas de « marcher » ; il marche en évitant de gaspiller les ressources de la machine. Sur certains programmes de grande taille, l'abus de variables surdimensionnées peut entraîner des ralentissements notables à l'exécution, voire un plantage pur et simple de l'ordinateur. Alors, autant prendre dès le début de bonnes habitudes d'hygiène.

En algorithmique, on ne se tracassera pas trop avec les sous-types de variables numériques (sachant qu'on aura toujours assez de soucis comme ça, allez). On se contentera donc de préciser qu'il s'agit d'un nombre, en gardant en tête que dans un vrai langage, il faudra être plus précis.

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

```
Variable g en Numérique
Ou encore
Variables PrixHT, TauxTVA, PrixTTC en Numérique
```

2.2 Autres types numériques

Certains langages autorisent d'autres types numériques, notamment :

- le type **monétaire** (avec strictement deux chiffres après la virgule)
- le type **date** (jour / mois / année).

Nous n'emploierons pas ces types dans ce cours ; mais je les signale, car vous ne manquerez pas de les rencontrer en programmation proprement dite.

2.3 Type alphanumérique

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

On dispose donc également du **type alphanumérique** (également appelé **type caractère**, **type chaîne** ou en anglais, le **type string** – mais ne fantasmez pas trop vite, c'est loin d'être aussi excitant que le nom le suggère…).

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

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

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

Pourquoi diable ? Pour éviter deux sources principales de possibles confusions :

- La confusion entre des nombres et des suites de chiffres. Par exemple, 423 peut représenter le nombre 423 (quatre cent vingt-trois), ou la suite de caractères 4, 2, et 3. Et ce n'est pas du tout la même chose ! Avec le premier, on peut faire des calculs, avec le second, point du tout. Dès lors, les guillemets permettent d'éviter toute ambiguïté : s'il n'y en a pas, 423 est quatre cent vingt trois. S'il y en a, "423" représente la suite des chiffres 4, 2, 3.
- …Mais ce n'est pas le pire. L'autre confusion, bien plus grave - et bien plus fréquente – consiste à se mélanger les pinceaux entre le nom d'une variable et son contenu. Pour parler simplement, cela consiste à confondre l'étiquette d'une boîte et ce qu'il y a à l'intérieur… On reviendra sur ce point crucial dans quelques instants.

2.4 Type booléen

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

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

> Le type booléen est très souvent négligé par les programmeurs, à tort.
>
> Il est vrai qu'il n'est pas à proprement parler indispensable, et qu'on pourrait écrire à peu près n'importe quel programme en l'ignorant complètement. Pourtant, si le type booléen est mis à disposition des programmeurs dans tous les langages, ce n'est pas pour rien. Le recours aux variables booléennes s'avère très souvent un puissant instrument de **lisibilité** des algorithmes : il peut faciliter la vie de celui qui écrit l'algorithme, comme de celui qui le relit pour le corriger.
>
> Alors, maintenant, c'est certain, en algorithmique, il y a une question de style : c'est exactement comme dans le langage courant, il y a plusieurs manières de s'exprimer pour dire sur le fond la même chose. Nous verrons plus loin différents exemples de variations stylistiques autour d'une même solution. En attendant, vous êtes prévenus : l'auteur de ce cours est un adepte fervent (mais pas irraisonné) de l'utilisation des variables booléennes.

3. L'instruction d'affectation

3.1 Syntaxe et signification

On aborde enfin nos premières véritables manipulations d'algorithmique. Pas trop tôt, certes, mais pas moyen de faire autrement !

En fait, la variable (la boîte) n'est pas un outil bien sorcier à manipuler. A la différence du couteau suisse ou du superbe robot ménager, on ne peut pas faire trente-six mille choses avec une variable, mais seulement une et une seule.

Cette seule chose qu'on puisse faire avec une variable, c'est **l'affecter,** c'est-à-dire **lui attribuer une valeur** ou encore on peut dire que l'on remplit la boîte.

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

Ainsi :

```
Toto ← 24
```

Attribue la valeur 24 à la variable Toto.

Ceci, soit dit en passant, sous-entend impérativement que Toto soit une variable de type numérique. Si Toto a été défini dans un autre type, il faut bien comprendre que cette instruction provoquera une erreur. C'est un peu comme si, en donnant un ordre à quelqu'un, on accolait un verbe et un complément incompatibles, du genre « Epluchez la casserole ». Même dotée de la meilleure volonté du monde, la ménagère lisant cette phrase ne pourrait qu'interrompre dubitativement sa tâche. Alors, un ordinateur, vous pensez bien…

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

```
Tutu ← Toto
```

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

Notez bien que cette instruction n'a en rien modifié la valeur de Toto : **une instruction d'affectation ne modifie que ce qui est situé à gauche de la flèche**.

```
Tutu ← Toto + 4
```

Si Toto contenait 12, Tutu vaut maintenant 16. De même que précédemment, Toto vaut toujours 12.

```
Tutu ← Tutu + 1
```

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

Pour revenir à présent sur le rôle des guillemets dans les chaînes de caractères et sur la confusion numéro 2 signalée plus haut, comparons maintenant deux algorithmes suivants :

Exemple n°1

```
Début
Riri ← "Loulou"
Fifi ← "Riri"
Fin
```

Exemple n°2

```
Début
 Riri ← "Loulou"
 Fifi ← Riri
Fin
```

La seule différence entre les deux algorithmes consiste dans la présence ou dans l'absence des guillemets lors de la seconde affectation. Et l'on voit que cela change tout !

Dans l'exemple n°1, ce que l'on affecte à la variable Fifi, c'est la suite de caractères R – i – r - i. Et à la fin de l'algorithme, le contenu de la variable Fifi est donc « Riri ».

Dans l'exemple n°2, en revanche, Riri étant dépourvu de guillemets, n'est pas considéré comme une suite de caractères, mais comme un nom de variable. Le sens de la ligne devient donc : « affecté à la variable Fifi le contenu de la variable Riri ». A la fin de l'algorithme n°2, la valeur de la variable Fifi est donc « Loulou ». Ici, l'oubli des guillemets conduit certes à un résultat, mais à un résultat différent.

A noter, car c'est un cas très fréquent, que généralement, lorsqu'on oublie les guillemets lors d'une affectation de chaîne, ce qui se trouve à droite du signe d'affectation ne correspond à aucune variable précédemment déclarée et affectée. Dans ce cas, l'oubli des guillemets se solde immédiatement par une erreur d'exécution.

Ceci est une simple illustration. Mais elle résume l'ensemble des problèmes qui surviennent lorsqu'on oublie la règle des guillemets aux chaînes de caractères.

3.2 Ordre des instructions

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

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

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

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

Il est tout aussi clair que ceci ne doit pas nous étonner. Lorsqu'on indique le chemin à quelqu'un, dire « prenez tout droit sur 1km, puis à droite » n'envoie pas les gens au même endroit que si l'on dit « prenez à droite puis tout droit pendant 1 km ».

Enfin, il est également clair que si l'on met de côté leur vertu pédagogique, les deux algorithmes ci-dessus sont parfaitement idiots. Malgré tout le moins ils contiennent une incohérence. Il n'y a aucun intérêt à affecter une variable pour l'affecter différemment juste après. En l'occurrence, on aurait tout aussi bien atteint le même résultat en écrivant simplement :

```
Variable A en Entier
Début
A ← 34
Fin
```

Tous les éléments sont maintenant en votre possession pour que ce soit à vous de jouer !

Exercice 1.1

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

```
Variables A, B en Entier
Début
A ← 1
B ← A + 3
A ← 3
Fin
```

Exercice 1.2

Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?

```
Variables A, B, C en Entier
Début
A ← 5
B ← 3
C ← A + B
A ← 2
C ← B - A
Fin
```

Exercice 1.3

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

```
Variables A, B en Entier
Début
A ← 5
B ← A + 4
A ← A + 1
B ← A - 4
Fin
```

Exercice 1.4

Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?

```
Variables A, B, C en Entier
Début
A ← 3
B ← 10
C ← A + B
B ← A + B
A ← C
Fin
```

Exercice 1.5

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

```
Variables A, B en Entier

Début
A ← 5
B ← 2
A ← B
B ← A
Fin
```

Moralité : les deux dernières instructions permettent-elles d'échanger les deux valeurs de B et A ? Si l'on inverse les deux dernières instructions, cela change-t-il quelque chose ?

Exercice 1.6

Plus difficile, mais c'est un classique absolu, qu'il faut absolument maîtriser : écrire un algorithme permettant d'échanger les valeurs de deux variables A et B, et ce, quel que soit leur contenu préalable.

Exercice 1.7

Une variante du précédent : on dispose de trois variables A, B et C. Ecrivez un algorithme transférant à B la valeur de A à C la valeur de B et à A la valeur de C (toujours quels que soient les contenus préalables de ces variables).

4. Expressions et opérateurs

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

- à gauche de la flèche, un nom de variable, et uniquement cela. En ce monde empli de doutes qu'est celui de l'algorithmique, c'est une des rares règles d'or qui marche à tous les

coups : si on voit à gauche d'une flèche d'affectation autre chose qu'un nom de variable, on peut être certain à 100% qu'il s'agit d'une erreur.

- à droite de la flèche, ce qu'on appelle une **expression**. Voilà encore un mot qui est trompeur ; en effet, ce mot existe dans le langage courant, où il revêt bien des significations. Mais en informatique, le terme d'**expression** ne désigne qu'une seule chose, et qui plus est, une chose très précise :

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

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

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

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

Revenons pour le moment sur l'affectation. Une condition supplémentaire (en plus des deux précédentes) de validité d'une instruction d'affectation est que :

- l'expression située à droite de la flèche soit du même type que la variable située à gauche. C'est très logique : on ne peut pas ranger convenablement des outils dans un sac à provision, ni des légumes dans une trousse à outils… sauf à provoquer un résultat catastrophique.

Si l'un des trois points énumérés ci-dessus n'est pas respecté, la machine sera incapable d'exécuter l'affectation, et déclenchera une erreur (est-il besoin de dire que si aucun de ces points n'est respecté, il y aura aussi erreur !)

On va maintenant détailler ce que l'on entend par le terme d'**opérateur**.

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

Les opérateurs possibles dépendent du type des valeurs qui sont en jeu. Allons-y, faisons le tour, c'est un peu fastidieux, mais comme dit le sage au petit scarabée, quand c'est fait, c'est plus à faire.

4.1 Opérateurs numériques :

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

+ addition
- soustraction
* multiplication
/ division

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

Enfin, on a le droit d'utiliser les parenthèses, avec les mêmes règles qu'en mathématiques. La multiplication et la division ont « naturellement » priorité sur l'addition et la soustraction. Les parenthèses ne sont ainsi utiles que pour modifier cette priorité naturelle. Cela signifie qu'en informatique, 12 * 3 + 5 et (12 * 3) + 5 valent strictement la même chose, à savoir 41. Pourquoi dès lors se fatiguer à mettre des parenthèses inutiles ?

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

4.2 Opérateur alphanumérique : &

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

Exemple

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

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

4.3 Opérateurs logiques (ou booléens) :

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

Exercice 1.8

Que produit l'algorithme suivant ?

```
Variables A, B, C en Caractères
Début
A ← "423"
B ← "12"
C ← A + B
Fin
```

Exercice 1.9

Que produit l'algorithme suivant ?

```
Variables A, B en Caractères
Début
A ← "423"
B ← "12"
C ← A & B
Fin
```

5. Deux remarques pour terminer

Maintenant que nous sommes familiers des variables et que nous les manipulons les yeux fermés (mais les neurones en éveil, toutefois), j'attire votre attention sur la trompeuse similitude de vocabulaire entre les mathématiques et l'informatique. En mathématiques, une « variable » est généralement une inconnue, qui recouvre un nombre non précisé de valeurs. Lorsque j'écris :

$y = 3x + 2$

les « variables » x et y satisfaisant à l'équation existent en nombre infini (graphiquement, l'ensemble des solutions à cette équation dessine une droite). Lorsque j'écris :

$ax^2 + bx + c = 0$

la « variable » x désigne les solutions à cette équation, c'est-à-dire zéro, une ou deux valeurs à la fois**…**

En informatique, une variable possède à un moment donné une valeur et une seule. A la rigueur, elle peut ne pas avoir de valeur du tout (une fois qu'elle a été déclarée, et tant qu'on ne l'a pas affectée. À signaler que dans certains langages, les variables non encore affectées sont considérées comme valant automatiquement zéro). Mais ce qui est important, c'est que cette valeur justement, ne « varie » pas à proprement parler. Du moins ne varie-t-elle que lorsqu'elle est l'objet d'une instruction d'affectation.

La deuxième remarque concerne le signe de l'affectation. En algorithmique, comme on l'a vu, c'est le ←. Mais en pratique, la quasi totalité des langages emploient le signe égal. Et là, pour les débutants, la confusion avec les maths est également facile. En maths, A = B et B = A sont deux propositions strictement équivalentes. En informatique, absolument pas, puisque cela revient à écrire A ← B et B ← A, deux choses bien différentes. De même, A = A + 1, qui en mathématiques, constitue une équation sans solution, représente en programmation une action tout à fait licite (et de surcroît extrêmement courante). Donc, attention !!! La meilleure des vaccinations contre cette confusion consiste à bien employer le signe ← en pseudo-code, signe qui a le mérite de ne pas laisser place à l'ambiguïté. Une fois acquis les bons réflexes avec ce signe, vous n'aurez plus aucune difficulté à passer au = des langages de programmation.

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'écriture

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…

Exercice 2.1

Quel résultat produit le programme suivant ?

```
Variables val, double numériques
Début
Val ← 231
Double ← Val * 2
Ecrire Val
Ecrire Double
Fin
```

Exercice 2.2

Ecrire un programme qui demande un nombre à l'utilisateur, puis qui calcule et affiche le carré de ce nombre.

Exercice 2.3

Ecrire un programme qui lit le prix HT d'un article, le nombre d'articles et le taux de TVA, et qui fournit le prix total TTC correspondant. Faire en sorte que des libellés apparaissent clairement.

Exercice 2.4

Ecrire un algorithme utilisant des variables de type chaîne de caractères, et affichant quatre variantes possibles de la célèbre « belle marquise, vos beaux yeux me font mourir d'amour ». On ne se soucie pas de la ponctuation, ni des majuscules.

Partie 3

LES TESTS

Je vous avais dit que l'algorithmique, c'est la combinaison de quatre structures élémentaires. Nous en avons déjà vu deux, voici la troisième. Autrement dit, on a quasiment fini le programme.

Mais non, je rigole.

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 là 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 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 forme de gauche est la forme complète, celle de droite la forme simple.

```
Si booléen Alors                    Si booléen Alors
      Instructions 1                    Instructions
Sinon                               Finsi
      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 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. 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 1. Cette série d'instructions peut être très brève comme très longue, cela n'a aucune importance. A la fin de cette série d'instructions, 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 ».

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.

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 TRES 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.

Exercice 3.1

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on laisse de côté le cas où le nombre vaut zéro).

4. Conditions composées

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.

- 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 est VRAIE. Le OU informatique ne veut donc pas dire « ou bien » (sauf dans certaines formes rares, dénommées OU exclusif et notées XOR).

- Le XOR (ou OU exclusif) fonctionne de la manière suivante. Pour que

  ```
  Condition1 XOR Condition2
  ```

  soit VRAI, il faut que Condition1 soit VRAI, ou bien que Condition2 soit VRAI. Mais si toutes les deux sont fausses, ou que toutes les deux sont VRAI, alors le résultat global est considéré comme FAUX.

  J'insiste 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.

  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, c'est évident qu'on se complique inutilement la vie avec le NON. Mais on rencontrera plus loin des situations dans lesquelles il rend de précieux services.

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 JOURNEE

Il 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.

Exercice 3.2

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leur produit est négatif ou positif (on laisse de côté le cas où le produit est nul). Attention toutefois : on ne doit **pas** calculer le produit des deux nombres.

Exercice 3.3

Ecrire un algorithme qui demande trois noms à l'utilisateur et l'informe ensuite s'ils sont rangés ou non dans l'ordre alphabétique.

5. Tests imbriqués

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 (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.

Exercice 3.4

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on inclut cette fois le traitement du cas où le nombre vaut zéro).

Exercice 3.5

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si le produit est négatif ou positif (on inclut cette fois le traitement du cas où le produit peut être nul). Attention toutefois, on ne doit pas calculer le produit !

Exercice 3.6

Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie :

- « Poussin » de 6 à 7 ans
- « Pupille » de 8 à 9 ans
- « Minime » de 10 à 11 ans
- « Cadet » après 12 ans

Peut-on concevoir plusieurs algorithmes équivalents menant à ce résultat ?

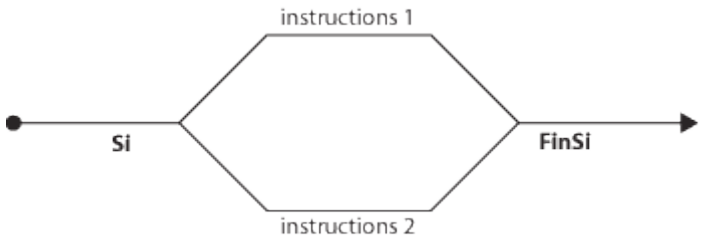
6. De l'aiguillage à la gare de tri

« 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.

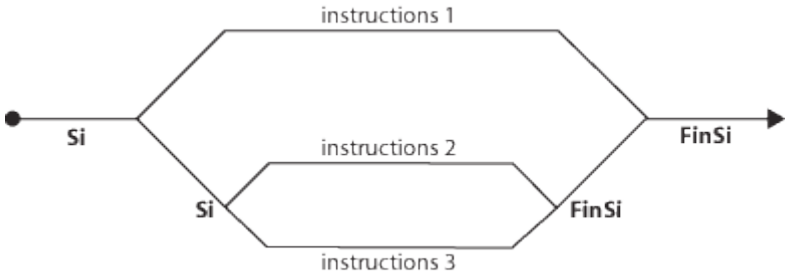
En effet, dans un programme, une structure SI peut être facilement comparée à un aiguillage de train. La voie principale se sépare en deux, le train devant rouler ou sur l'une, ou sur l'autre, et

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 Booléennes

Jusqu'ici, nous avons utilisé uniquement des **conditions** pour faire des tests. 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 peut 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 < 2
B ← X > 12
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                    Si NON A OU NON B Alors
     Instructions 1                          Instructions 2
Sinon                        ⇔     Sinon
     Instructions 2                          Instructions 1
Finsi                              Finsi
```

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

Exercice 4.1

Formulez un algorithme équivalent à l'algorithme suivant :

```
Si Tutu > Toto + 4  OU Tata = "OK" Alors
     Tutu ⟵ Tutu + 1
Sinon
     Tutu ⟵ Tutu – 1
Finsi
```

Exercice 4.2

Cet algorithme est destiné à prédire l'avenir, et il doit être infaillible !

Il lira au clavier l'heure et les minutes, et il affichera l'heure qu'il sera une minute plus tard. Par exemple, si l'utilisateur tape 21 puis 32, l'algorithme doit répondre : "Dans une minute, il sera 21 heure(s) 33".

NB : on suppose que l'utilisateur entre une heure valide. Pas besoin donc de la vérifier.

Exercice 4.3

De même que le précédent, cet algorithme doit demander une heure et en afficher une autre. Mais cette fois, il doit gérer également les secondes, et afficher l'heure qu'il sera une seconde plus tard.

Par exemple, si l'utilisateur tape 21, puis 32, puis 8, l'algorithme doit répondre : "Dans une seconde, il sera 21 heure(s), 32 minute(s) et 9 seconde(s)".

NB : là encore, on suppose que l'utilisateur entre une date valide.

Exercice 4.4

Un magasin de reprographie facture 0,10 E les dix premières photocopies, 0,09 E les vingt suivantes et 0,08 E au-delà.

Ecrivez un algorithme qui demande à l'utilisateur le nombre de photocopies effectuées et qui affiche la facture correspondante.

Exercice 4.5

Les habitants de Zorglub paient l'impôt selon les règles suivantes :

· les hommes de plus de 20 ans paient l'impôt

· les femmes paient l'impôt si elles ont entre 18 et 35 ans

· les autres ne paient pas d'impôt

Le programme demandera donc l'âge et le sexe du Zorglubien, et se prononcera donc ensuite sur le fait que l'habitant est imposable.

2. Au-delà 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.

Exercice 4.6

Les élections législatives, en Guignolerie Septentrionale, obéissent à la règle suivante :

- lorsque l'un des candidats obtient plus de 50% des suffrages, il est élu dès le premier tour.
- en cas de deuxième tour, peuvent participer uniquement les candidats ayant obtenu au moins 12,5% des voix au premier tour.

Vous devez écrire un algorithme qui permette la saisie des scores de quatre candidats au premier tour. Cet algorithme traitera ensuite le candidat numéro 1 (et **uniquement** lui) : il dira s'il est élu, battu, s'il se trouve en ballottage favorable (il participe au second tour en étant arrivé en tête à l'issue du premier tour) ou défavorable (il participe au second tour sans avoir été en tête au premier tour).

Exercice 4.7

Une compagnie d'assurance automobile propose à ses clients quatre familles de tarifs identifiables par une couleur, du moins au plus onéreux : tarifs bleu, vert, orange et rouge.

Le tarif dépend de la situation du conducteur :

- un conducteur de moins de 25 ans et titulaire du permis depuis moins de deux ans, se voit attribuer le tarif rouge, si toutefois il n'a jamais été responsable d'accident. Sinon, la compagnie refuse de l'assurer.

- un conducteur de moins de 25 ans et titulaire du permis depuis plus de deux ans, ou de plus de 25 ans mais titulaire du permis depuis moins de deux ans a le droit au tarif orange s'il n'a jamais provoqué d'accident, au tarif rouge pour un accident, sinon il est refusé.

- un conducteur de plus de 25 ans titulaire du permis depuis plus de deux ans bénéficie du tarif vert s'il n'est à l'origine d'aucun accident et du tarif orange pour un accident, du tarif rouge pour deux accidents, et refusé au-delà

De plus, pour encourager la fidélité des clients acceptés, la compagnie propose un contrat de la couleur immédiatement la plus avantageuse s'il est entré dans la maison depuis plus d'un an.

Ecrire l'algorithme permettant de saisir les données nécessaires (sans contrôle de saisie) et de traiter ce problème. Avant de se lancer à corps perdu dans cet exercice, on pourra réfléchir un peu et s'apercevoir qu'ils est plus simple qu'il en a l'air (cela s'appelle faire une analyse !)

Exercice 4.8

Ecrivez un algorithme qui a près avoir demandé un numéro de jour, de mois et d'année à l'utilisateur, renvoie s'il s'agit ou non d'une date valide.

Cet exercice est certes d'un manque d'originalité affligeant, mais après tout, en algorithmique comme ailleurs, il faut connaître ses classiques ! Et quand on a fait cela une fois dans sa vie, on apprécie pleinement l'existence d'un type numérique « date » dans certains langages…).

Il n'est sans doute pas inutile de rappeler rapidement que le mois de février compte 28 jours, sauf si l'année est bissextile, auquel cas il en compte 29. L'année est bissextile si elle est divisible par quatre. Toutefois, les années divisibles par 100 ne sont pas bissextiles, mais les années divisibles par 400 le sont. Ouf !

Un dernier petit détail : vous ne savez pas, pour l'instant, exprimer correctement en pseudo-code l'idée qu'un nombre A est divisible par un nombre B. Aussi, vous vous contenterez d'écrire en bons télégraphistes que A divisible par B se dit « A dp B ».

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 : ce sont 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 est 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 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
```

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

Exercice 5.1

Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

Exercice 5.2

Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus grand ! » si le nombre est inférieur à 10.

Exercice 5.3

Ecrire un algorithme qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres de 18 à 27.

LE GAG DE LA JOURNEE

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.

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).

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 (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 on 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 réciproquement* »)

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. N'y a pas de raison.

```
Variables Truc, Trac en Entier
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
```

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
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
```

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 bêtise à ne pas faire !

Examinons l'algorithme suivant :

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

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 JOURNEE

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.

Exercice 5.4

Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur entre le nombre 7) :

```
Table de 7 :

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
…
7 x 10 = 70
```

Exercice 5.5

Ecrire un algorithme qui demande un nombre de départ, et qui calcule la somme des entiers jusqu'à ce nombre. Par exemple, si l'on entre 5, le programme doit calculer :

```
1 + 2 + 3 + 4 + 5 = 15
```

Exercice 5.6

Ecrire un algorithme qui demande un nombre de départ, et qui calcule sa factorielle.

NB : la factorielle de 8, notée 8 !, vaut 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8

Exercice 5.7

Ecrire un algorithme qui demande successivement 20 nombres à l'utilisateur, et qui lui dise ensuite quel était le plus grand parmi ces 20 nombres :

```
Entrez le nombre numéro 1 : 12
Entrez le nombre numéro 2 : 14
```
Etc.
```
Entrez le nombre numéro 20 : 6

Le plus grand de ces nombres est  : 14
```

Modifiez ensuite l'algorithme pour que le programme affiche de surcroît en quelle position avait été saisie ce nombre :

```
C'était le nombre numéro 2
```

Exercice 5.8

Réécrire l'algorithme précédent, mais cette fois-ci on ne connaît pas d'avance combien l'utilisateur souhaite saisir de nombres. La saisie des nombres s'arrête lorsque l'utilisateur entre un zéro.

Exercice 5.9

Lire la suite des prix (en euros entiers et terminée par zéro) des achats d'un client. Calculer la somme qu'il doit, lire la somme qu'il paye, et simuler la remise de la monnaie en affichant les textes « 10 E », « 5 E » et « 1 E » autant de fois qu'il y a de coupures de chaque sorte à rendre.

Exercice 5.10

Ecrire un algorithme qui permette de connaître ses chances de gagner au tiercé, quarté, quinté et autres impôts volontaires.

On demande à l'utilisateur le nombre de chevaux partants, et le nombre de chevaux joués.
Les deux messages affichés devront être :

Dans l'ordre : une chance sur X de gagner
Dans le désordre : une chance sur Y de gagner

X et Y nous sont donnés par la formule suivante, si n est le nombre de chevaux partants et p le nombre de chevaux joués :

X = n ! / (n - p) !
Y = n ! / (p ! * (n – p) !)

NB : Cet algorithme peut être écrit d'une manière simple, mais relativement peu performante. Ses performances peuvent être singulièrement augmentées par une petite astuce. Vous commencerez par écrire la manière la plus simple, puis vous identifierez le problème, et écrirez une deuxième version permettant de le résoudre.

Programmation Langage C

1/ Introduction.

La programmation est le moyen de créer sur un ordinateur une très grande variété d'outils informatiques, c'est-à-dire des logiciels. Un programme n'est rien d'autre qu'un enchaînement d'opérations élémentaires permettant d'atteindre un résultat précis à partir de données précises.

Pour réaliser un programme qui fonctionne correctement, il faut donc :

- déterminer les données du programme (c'est-à-dire les informations sur lesquelles le programme va être exécuté)
- déterminer le résultat attendu (c'est-à-dire le but du programme)
- décider des opérations à utiliser et les ordonner afin que les données soient traitées de telle manière qu'on obtienne systématiquement le résultat attendu (c'est le principe de l'algorithmie)
- s'assurer que chaque instruction respecte rigoureusement sa syntaxe
- tester le programme et corriger toutes les erreurs de syntaxe et de logique

1.1) Les données et les résultats.

Une des premières difficultés que rencontre un débutant en programmation ou en algorithmie, c'est de décider d'où vont provenir les données de départ, et que va-t-on faire des résultats ?

Les données ont deux origines possibles : soit elles proviennent de l'utilisateur du programme (notion de communication), soit elles proviennent d'une autre partie du programme (notion d'archivage). Ces deux notions sont à maîtriser absolument.

Exemple : un professeur souhaite créer un programme qui calcule la moyenne de ses élèves en fonction de leurs notes. Ici, le but recherché, c'est-à-dire le résultat, c'est la moyenne de chaque élève. Quant aux données de départ, c'est-à-dire ce qui permettra de calculer cette moyenne, ce sera les notes de chaque élève. Le professeur a donc deux possibilités pour introduire les notes dans le programme. Première possibilité : il donne les notes au programme en les tapant au clavier pendant que celui-ci est en cours d'exécution. C'est la technique dite de « communication » (l'utilisateur va communiquer les données au programme). Deuxième possibilité : il enregistre préalablement les notes des élèves dans la mémoire ou sur un disque de l'ordinateur, puis il exécute le programme qui va aller chercher ses données là où elles ont été stockées. C'est la technique dite d'« archivage » (les données ont été archivées dans la mémoire ou sur le disque).

Généralement, la technique de l'archivage est inévitablement utilisée quel que soit le programme que l'on écrit. En effet, dès que l'on utilise la moindre variable, on fait de l'archivage puisqu'on met une valeur dans la partie de la mémoire allouée à cette variable. D'autre part, il est quasiment inévitable de faire suivre une communication d'un archivage, car il faut bien conserver les données que l'utilisateur a données au programme.

A la base, l'archivage se fait avec des variables auxquelles on affecte une valeur, mais on verra qu'il existe d'autres façons d'archiver des données, par exemple en les inscrivants dans un fichier (c'est-à-dire non plus dans la mémoire de l'ordinateur, mais sur un disque).

Puis va venir la question concernant les résultats du programme : que va-t-on en faire ? Là encore, la question se pose en terme de communication ou d'archivage : va-t-on communiquer les données à l'utilisateur (en les affichant, par exemple) ? Va-t-on les conserver dans un fichier pour une utilisation ultérieur ? Va-t-on les mettre à disposition pour que d'autres programmes puissent s'en servir ?

1.2) Le codage.

On sait que l'ordinateur code toutes les données sous la forme d'enchaînement de « 0 » et de « 1 ». Chaque valeur (0 ou 1) est placée dans une cellule de la mémoire qu'on appelle un « bit ». En rassemblant les bits par 4, on forme des quartets, et par 8, on forme des octets. Cette façon est la plus courante de comptabiliser l'espace mémoire occupée par une donnée. Mais il y a une notion très importante à bien comprendre dans le codage de l'ordinateur : un code seul représente une valeur, mais on ne sait pas quel est le type de cette valeur (nombre entier, nombre flottant, caractère, booléen, image, son, instruction de programme, etc. ?). C'est pour cette raison que la programmation nécessite une grande attention dans la manipulation des types des données, qui est d'ailleurs une grande source d'erreurs de logique.

D'autre part, il est bon de rappeler qu'un bit ne peut contenir que 0 OU 1. Il ne peut contenir les deux en même temps et il ne peut pas ne rien contenir. Par conséquent, tout ce qui fait référence à une partie de la mémoire ne peut pas être vide (une variable, par exemple, contient obligatoirement une valeur, même si on ne lui a rien affecté, d'où un risque d'erreur fréquent chez les débutants en programmation).

1.3) Le compilateur.

On appelle compilateur le logiciel qui traduit les algorithmes écrits en langage C en langage machine. D'une certaine manière, on peut dire qu'il rend compréhensible pour l'ordinateur les programmes écrits par un programmeur. Généralement, on lance le compilateur sur un programme pour vérifier si l'ordinateur le comprend ou s'il repère des erreurs. Attention, l'ordinateur n'est pas capable de repérer toutes les erreurs d'un programme, mais il s'aperçoit toujours des erreurs de syntaxe, car ces erreurs rendent le programme inexploitable pour lui. D'autre part, il peut également signaler tous détails qui lui paraissent douteux au moment de la compilation. C'est ce qu'on appelle des « warnings » (ou avertissements).

1.4) Comment exécuter un programme avec Dev-C++ ?

Voici une façon de faire :

Lancer le logiciel Dev-C++

Ouvrir le fichier source d'un programme existant ou créer un nouveau fichier source et y écrire un programme

Ici donner l'exemple du programme « hello world » :

HELLO_WORLD en langage algorithmique

```
programme HELLO_WORLD
| (* Notre premier programme en C *)
```

```
  |   écrire "hello, world"
  fprogramme
```

HELLO_WORLD en C

```
  #include <stdio.h>
  main()
  /* Notre premier programme en C */
  {
    printf("hello, world\n");
    return 0;
  }
```

Lancer le compilateur pour vérifier la présence d'éventuelles erreurs (s'il n'y a pas d'erreur, le compilateur va créer un fichier exécutable de votre programme dans le répertoire de votre fichier source)

Corriger les éventuelles erreurs

Dans le menu Start, cliquez sur « Run »

Taper « command » et cliquer sur « Ok », ce qui ouvrira une fenêtre DOS

Ouvrir le répertoire dans lequel votre fichier source a été enregistré, avec la commande « cd »

Redonner les quelques commandes DOS si nécessaire. (dir, cd, …)

Lancer l'exécutable créé à partir de votre fichier source : c'est là que votre programme s'exécute réellement

1.5) Comment obtenir de l'aide lorsqu'on programme en C ?

Tout d'abord, le logiciel de développement utilisé possède probablement des fichiers d'aide généralement accessibles par le menu « Aide » ou « Help ». Dev-C++ contient effectivement ce genre de fichier, mais cette documentation est en Anglais et relativement incomplète. Par contre TcLite contient, lui des fichiers d'aide très précis, en particulier pour tout ce qui concerne l'utilisation des fonctions prédéfinies. Il suffit donc de lancer TcLite pour accéder à l'index dans le menu « Help » et y retrouver la page d'explication de chaque fonction prédéfinie du C (en Anglais également). Enfin, dans le cas où ces fichiers n'apportent pas de réponse satisfaisante, il existe sur Internet divers moyens de trouver de l'aide auprès des communautés virtuelles de développeurs en C.

2/ Les variables.

Comme en algorithmie, tout le travail de la programmation s'articule autour des notions de variable et de type. On sait qu'une variable est identifiée par son nom et définie par son type et sa valeur.

2.1) Le nom.

Il est laissé au choix du programmeur au moment de la déclaration, mais avec les contraintes suivantes :

les caractères autorisés sont les suivants : toutes les lettres minuscules, toutes les lettres majuscules, tous les chiffres et le symbole « _ ». Les lettres accentuées (é, è, à, etc…) ou modifiées (ç, ï, etc…) ne sont pas admises.

le premier caractère ne doit pas être un chiffre.

un nom ne doit pas comporter plus de 32 caractères (si on le fait, le compilateur l'accepte, mais sans tenir compte des caractères superflus, d'où un risque d'erreurs).

aucun espace n'est admis.

le langage C respecte la casse, c'est-à-dire que les lettres en minuscules sont différentes des mêmes lettres en majuscules.

Exemples

Identificateurs corrects:	Identificateurs incorrects:
nom1	1nom
nom_2	nom.2
_nom_3	-nom-3
Nom_de_variable	Nom de variable
deuxieme_choix	deuxième_choix
mot_francais	mot_français

2.2) Le type.

Le type d'une variable précise l'ensemble des valeurs qu'elle peut contenir, la taille de son emplacement dans la mémoire de l'ordinateur, la manière dont l'information y est codée et les opérations autorisées.

Les premiers types de variables disponibles en C sont les suivants :

- int : nombres entiers signés (autrement dit, ce sont les nombres relatifs)
- short : nombres entiers signés codés sur un nombre plus restreint de bits que le type int
- long : nombres entiers signés codés sur un plus grand nombre de bits que le type int
- float : nombres à virgule flottante (approximation des nombres réels)
- double : nombres à virgule flottante codés sur un plus grand nombre de bits que le type float
- long double : nombres à virgule flottante codés sur un plus grand nombre de bits que le type double

- char : caractères

De plus, les types int, short et long peuvent être précédés du mot-clef « unsigned » qui supprime la prise en charge des valeurs négatives. Ainsi, on dispose d'une marge deux fois plus grande pour les valeurs positives.

D'autre part, on peut remarquer qu'il n'y a pas de type booléen. En fait, en C, les variables booléennes doivent être simulées avec une variable d'un autre type dont on ne considérera que deux valeurs possibles. Sachez cependant que le type booléen a été introduit dans les compilateurs C, mais il est préférable de savoir simuler ce type à l'aide d'une variable d'un autre type.

Le nombre de bits utilisés pour le codage dépend de la puissance de l'ordinateur utilisé pour compiler et exécuter le programme, et il peut arriver que deux types proches comme short et int ou comme float et double, disposent du même nombre de bits sur certaines machines.

2.3) Instruction de déclaration d'une variable.

Comme en algorithmie, les instructions de déclaration précèdent les autres instructions.

Syntaxe : <type> <nom> ;

Exemple : int X ;

Ici, on a déclaré une variable de type « int » nommée X.

La plupart des instructions du langage C sont terminées par un point virgule. L'oublie de ce symbole engendre inévitablement une erreur de syntaxe lors de la compilation. Par contre, grâce à cette contrainte syntaxique, il est tout à fait possible de mettre autant d'instructions que l'on veut sur la même ligne.

Egalement comme en algorithmie, il est tout à fait possible de déclarer plusieurs variables de même type dans la même instruction en les séparant par une virgule. Exemple : int X, Y, Z ;

2.4) Valeur d'une variable.

En algorithmie, on insiste sur le fait qu'une variable ne reçoit AUCUNE valeur lors de sa déclaration, et doit donc recevoir une première valeur avant de pouvoir participer à une autre opération. Dans un ordinateur, on a rappelé précédemment que les bits qui servent au codage ne peuvent pas ne rien contenir. Par conséquent, en programmation, puisque lors de la déclaration d'une variable, le compilateur lui alloue une portion de mémoire qui doit contenir sa valeur, ces quelques bits contiennent donc forcément déjà le code d'une valeur. Donc, lors de sa déclaration, une variable reçoit forcément une valeur, mais une valeur complètement imprévisible. Il est important de bien comprendre cet aspect, car si on oublie d'initialiser une variable, on va travailler sur une valeur aléatoire, ce qui fait que l'on obtiendra forcément un résultat également aléatoire, d'où un important risque d'erreur de logique.

Pour éviter ce genre d'erreur, le langage C autorise l'initialisation des variables à l'intérieur de l'instruction de déclaration en en complétant la syntaxe de la façon suivante :

<type> <nom> = <valeur d'initialisation> ;

Exemple : int X = 5 ;

Ici, on a déclaré une variable de type « int » nommée X et à laquelle on donne comme première valeur 5.

Au cours du reste du programme, l'affectation de valeur aux variables se fait avec les mêmes règles qu'en algorithmie, selon la syntaxe suivante :

<nom de la variable> = <valeur> ;

Notons ici encore la présence impérative du point-virgule pour fermer l'instruction. De façon générale, on peut dire que l'instruction d'affectation a pour rôle de calculer la valeur de l'expression figurant à droite du signe « = » et de ranger cette valeur dans la variable placée à gauche de « = ».

3/ Instructions et expressions.

Une instruction est un ordre donné à l'ordinateur selon une syntaxe de programmation. Un programme est simplement un enchaînement ordonné d'instructions. Une instruction est composée d'opérateurs et d'opérandes (valeurs constantes, variables ou expressions elles-mêmes composées d'opérateurs et d'opérandes). Il est important de bien distinguer l'instruction complète des expressions qui la composent. Une expression a souvent un type et une valeur qu'il faut savoir déterminer pour prévenir ou corriger les erreurs de calcul ou de conversion de type.

Exemple : X = Y + 5 est une instruction d'affectation composée d'une variable (X), de l'opérateur d'affectation (=) et d'une expression (Y + 5).

3.1) Les opérateurs mathématiques.

Les opérations mathématiques de base s'effectuent avec les symboles suivants :

- addition : +
- soustraction et opposé : -
- multiplication : *
- division : /

Le langage C propose également un autre opérateur appelé « modulo », correspondant au reste de la division entière. Son symbole est : « % ».

Exemple : 11%4 vaut 3, car la division de 11 par 4 a pour reste 3.

Lorsque plusieurs opérateurs mathématiques apparaissent dans la même expression, les règles de priorité sont les même qu'en algèbre et les parenthèses peuvent être utilisées de la même manière.

Exemple : dans l'opération A + B * C, on multiplie d'abord B par C, puis on ajoute A. De plus, l'opérateur % a la même priorité que la multiplication.

3.2) Dépassement des limites du codage.

Nous savons que le codage des variables apportent une limite aux valeurs qu'elles peuvent contenir (par exemple, une variable de type int codée sur 2 octets ne peut pas contenir de valeur inférieur à –32768 ou supérieur à 32767). Par conséquent, il se peut que par exemple, une variable de type int reçoive le résultat d'un calcul supérieur à la limite du type int. Dans ce cas,

la plupart du temps, on obtiendra un résultat aberrant, car les données excédentaires seront tout simplement ignorées, et ce sans aucun avertissement de la part du compilateur.

Par contre, si on effectue une opération non conforme (comme par exemple, une division par 0), normalement le compilateur doit bloquer l'exécution du programme à ce niveau en affichant un message d'erreur.

3.3) Incrémentation et décrémentation unitaire.

Pour l'incrémentation et la décrémentation unitaire d'une variable de type numérique, les instructions utilisées en algorithmie sont basées sur le schéma de l'exemple suivant : X <-X + 1 ou X <- X – 1. En C, il existe une instruction spéciale pour l'incrémentation et la décrémentation unitaire, dont voici les différentes syntaxes :

Incrémentation : ++ <variable> ou <variable> ++

Décrémentation : -- <variable> ou <variable> --

Exemple : l'opération X++ ; est entièrement équivalente à celle-ci : X = X + 1 ;

Ces deux syntaxes ne présentent aucune différence si on les utilise seules en tant qu'instruction. Ainsi, les instructions X++ ; et ++X ; sont équivalentes. Par contre, si on utilise ces syntaxes en tant qu'expression à l'intérieur d'une autre instruction, on doit prendre en compte la distinction suivante :

++ <variable> ou -- <variable> : la valeur de cette expression est celle de la variable après l'incrémentation ou la décrémentation.

<variable> ++ ou <variable> -- : la valeur de cette expression est celle de la variable avant l'incrémentation ou la décrémentation.

De la même manière, il existe en C pour chaque opération mathématique, un opérateur rassemblant l'affectation et l'opération mathématique. Exemples :

X = X + K ; peut être remplacé par X += K ;

Y = Y * X ; peut être remplacé par Y *= X ;

3.4) Particularités des types flottant.

Les valeurs de type float peuvent s'écrire de deux façons différentes :

- notation décimale (exemples : 3.5 9. .7 -4. -.6)
- notation exponentielle (exemples : 4E3 4.25E7 3E-2 -9.4E6)

Le symbole modulo n'est pas utilisable avec les valeurs de type float.

3.5) Expressions simples et expressions mixtes.

Une expression simple est une opération ne contenant qu'un seul type de valeurs ou de variables (exemple : l'instruction X = X * 2 ; où X est une variable de type int, est une expression simple). Par opposition, les expressions mixtes sont des opérations dans lesquelles on fait intervenir des valeurs de type différent, en particulier int et float. Au départ, ce genre

d'opération est impossible à réaliser à cause des différences de codage, mais le compilateur réalise automatiquement des conversions de type afin de rendre possible les calculs. Ces conversions de type sont similaires à celles étudiées en algorithmie, et il est important de bien en tenir compte car elles peuvent être à l'origine d'une perte de données au niveau du résultat.

Les conversions de type sont réalisées au fur et à mesure des besoins, en suivant les règles de priorité habituelles dans les opérations contenant plusieurs opérateurs. De façon générale, on parle du type d'une expression pour désigner le type du résultat d'une opération.

Exercice : soient X et Y de type int, et A et B de type float. Donner le type des expressions suivantes :

Corrigé :

X + Y	int
A * B	float
X / Y	float
X + A	float
X + B * A	float
X + Y * B	float
X + 5	int
X + 2.3	float
Y * 3	int
B / 2	float

Une conversion de type a également lieue lors d'une affectation si la valeur affectée est de type différent que la variable qui la reçoit. Là encore, il faut donc y être attentif, car cela peut produire des effets inattendus si la conversion est faite involontairement.

3.6) Conversions forcées.

Il est tout à fait possible de forcer la conversion d'une variable à l'intérieur d'une expression de manière à éviter de perdre des données. Il existe un opérateur de conversion forcée (appelé opérateur de « cast ») pour chaque type. En voici la syntaxe : (<Type>)

Par exemple, dans le programme suivant, la division est de type int et on risque donc d'obtenir un résultat imprécis :

```
{
int n=5, p=9;
float x;
```

```
x = p / n;
printf ("%f\n",x);        /* Ici, on obtiendra donc 1 comme résultat */
}
```

Pour éviter cette perte de précision, on va forcer le type de l'expression en float en ajoutant ce type entre parenthèses à l'intérieur de l'expression elle-même :

```
{
int n=5, p=9;
float x;
x = (float) p / n;
printf ("%f\n",x);        /* Ici, on obtiendra 1.8 comme résultat */
}
```

3.7) Le type char.

Ce type correspond au type « car » utilisé en algorithmie pour les caractères. Les valeurs constantes doivent impérativement est placées entre les symboles ' et ' faute de quoi elles seraient considérées par le compilateur comme des variables et non comme des valeurs de type char.

Exemple : lorsqu'on écrit l'instruction « a = 'e'; » , cela signifie que la variable a reçoit la valeur « e », alors que l'instruction « a = e ; » signifie que la variable a reçoit la valeur de la variable e.

Il existe en informatique de nombreux caractères non imprimables comme les sauts de ligne, la tabulation, le retour arrière ou l'activation de l'alarme sonore. Ces caractères nécessitent une syntaxe particulière pour pouvoir être utilisés. De plus, certains autres caractères classiques nécessitent également une notation spéciale compte tenu de leur rôle particulier. Voici les exemples les plus utilisés :

'\n' : saut de ligne

'\b' : retour arrière

'\a' : cloche ou bip (ce n'est pas réellement un caractère, mais en fait un son qui est joué lorsque le compilateur rencontre ce symbole)

'\\' : caractère \

'\'' : caractère '

'\''' : caractère ''

Conversion int / char et char / int :

Lorsque le compilateur doit convertir une valeur numérique et valeur alpha-numérique ou inversement, il interprète simplement le code binaire de valeur dans un type différent. Ainsi, on peut remarquer que chaque lettre de l'alphabet possède une correspondance de type int. Ainsi, il est tout à fait possible d'effectuer des opérations arithmétiques sur des valeurs ou des variables de type char qui seront d'abord converties au type int.

4/ Communication entre le programme et l'utilisateur.

Un programme est capable de communiquer à travers tous les périphériques dont dispose un ordinateur de manière à obtenir les données dont il a besoin et à transmettre les résultats qu'il trouve. Dans un premier temps, nous nous limiterons aux deux périphériques de communication avec l'utilisateur les plus courants : l'écran et le clavier.

4.1) Affichage à l'écran.

L'affichage de texte et de valeurs se fait grâce à la fonction printf. Cette fonction est légèrement plus complexe que la fonction « Afficher » utilisée en algorithmie, mais elle présente le gros avantage d'accepter autant de paramètres qu'on le souhaite. L'affichage d'un simple texte sans aucune variable utilise la syntaxe suivante :

printf (« <texte> ») ;

Par exemple, l'instruction suivante affiche le message « bonjour » : printf (« Bonjour ») ;

Généralement, on appelle « libellé » le texte affiché avec la fonction printf.

L'affichage de valeurs des variables et des expressions se fait en deux temps : d'abord indiquer l'emplacement et le type de la variable ou de l'expression dans la partie qu'on appellera « format d'affichage », ensuite donner la variable ou l'expression elle-même dans la partie qu'on appellera « liste ». Le type est indiqué grâce aux codes de format suivants :

%d : types int, short et long

%c : type char

%e : types float, double et long double sous la forme exponentielle

%f : types float, double et long double sous la forme décimale

Pour indiquer l'emplacement où sera affichée la valeur de la variable, il suffit de placer le code de format correspondant à l'endroit voulu.

Exemples :

On souhaite afficher la valeur de la variable X de type int. Ici, l'emplacement ne posera aucune difficulté puisque la variable n'est pas accompagnée de texte.

printf (« %d », X) ;

On souhaite afficher la valeur des variables X, Y et Z toutes de type int. Ici, pour ce qui est de l'emplacement, on tient à afficher d'abord X, puis Y et enfin Z.

printf (« %d » « %d » « %d », X, Y, Z) ;

On souhaite afficher la valeur de la variable X de type int, précédée du texte « Voici la valeur de X : ». Ici, pour ce qui est de l'emplacement, on souhaite que la valeur soit affichée à un endroit précis pas rapport au texte.

printf (« Voici la valeur de X : %d », X) ;

La fonction printf permet également d'afficher le résultat d'une ou plusieurs expressions. Dans ce cas, il faut bien déterminer le type de l'expression pour pouvoir décider du code de format que l'on va utiliser. Exemple :

pritnf (« %d », X+ Y) ;

Maintenant que l'on sait demander à l'ordinateur d'afficher la valeur des variables, on peut facilement tester le résultat de différentes expressions contenant des conversions de type.

4.2) Notions de gabarit et de précision d'affichage.

Dans le code de format d'une valeur numérique, il est possible d'introduire un nombre après le symbole « % » afin de préciser le nombre minimal de caractères à utiliser pour l'affichage de la valeur correspondante. Si la valeur peut s'écrire avec moins de caractères, elle sera alors précédée du nombre suffisant d'espaces.

Les valeurs de type float sont affichées par défaut avec 6 chiffres après la virgule. Il est possible de modifier cette précision en indiquant le nombre de chiffres souhaités, précédé d'un point, après le symbole « % » et le gabarit si celui-ci est présent.

Exemples :

x = 1.2345 ;

printf (« %10.3f », x) ; On obtient : « 1.235»

x = 1.2345E3 ;

printf (« %10.3f », x) ; On obtient : « 1234.500»

x = 1.2345E7 ;

printf (« %10.3f », x) ; On obtient : «12345000.000»

x = 1.2345 ;

printf (« %.2f », x) ; On obtient : «1.24»

Une autre façon d'améliorer le gabarit d'affichage consiste à placer des sauts de ligne grâce au symbole « \n » de manière à améliorer la présentation des données affichées.

4.3) Lecture au clavier.

La lecture des valeurs tapées au clavier par l'utilisateur se fait avec la fonction scanf. Son rôle est de recevoir la valeur venant du clavier, de la convertir dans le type voulu et de la placer dans la variable qui lui a été assignée. Voici sa syntaxe :

scanf (« <type de la valeur lue> », &< variable d'accueil de la valeur lue>) ;

Exemple : scanf (« %d », &X) ; Cette instruction lit au clavier une valeur de type int et l'affecte à la variable X.

Lorsqu'on veut lire plusieurs variables dans une seule instruction, il suffit de répéter la syntaxe autant de fois que nécessaire. Exemple :

scanf (« %d%d%d », &X, &Y, &Z) ; Cette instruction lit au clavier trois valeurs de type int et les affecte successivement aux variables X, Y et Z.

Au moment de l'exécution du programme, si jamais l'utilisateur ne fournit pas suffisament de données, le programme restera en attente jusqu'à ce que toutes les données attendues soient fournies par l'utilisateur. D'autre part, si l'utilisateur fournit trop de données, les données en trop seront conservées pour la lecture suivante ou abandonnées si le programme ne contient aucune autre instruction de lecture.

En ce qui concerne le symbole « & », il signifie que scanf n'attend pas le nom d'une variable, mais son adresse mémoire. Cette notion sera étudiée plus tard, mais notons que ce symbole est ici absolument obligatoire. D'autre part, ici la notion de libellé n'a aucune raison d'être, de même que la liste ne contiendra que des variables et non plus des expressions.

La lecture au clavier lors de l'exécution du programme est assez naturelle : l'utilisateur tape sa valeur et la valide avec la touche « Entrée ». Tant qu'il ne l'a pas validée, il peut la modifier à volonté. Lorsqu'il y a plusieurs valeurs à lire, l'utilisateur peut passer de l'une à l'autre avec les touches « Entrée », « Espace » ou « Tabulation », mais la dernière doit forcément être validée par la touche « Entrée ».

En ce qui concerne la lecture des valeurs de type flottant, les codes de format %e et %f sont équivalents et les valeurs peuvent être lues sous n'importe quelle forme.

Pour la lecture des valeurs de type char, on rencontre certaines difficultés lorsque l'on souhaite en lire plusieurs successivement : en effet, ici les caractères « Espace », « Tabulation » ou le saut de ligne sont considérés comme des séparateurs et en même temps comme des caractères. Par conséquent, au moment de la lecture, lorsque l'utilisateur tape par exemple la touche Entrée pour valider sa saisie, le caractère « saut de ligne » devient disponible pour la lecture d'un caractère.

Exemple : scanf (« %c%c »,&C1, &C2) ;

1er cas : l'utilisateur peut taper la réponse suivante : ab, puis valider par la touche Entrée. Dans ce cas, C1 va recevoir 'a', C2 v recevoir 'b', mais le caractère « saut de ligne » restera automatiquement disponible pour la prochaine lecture. Donc, si cette prochaine lecture porte sur une variable de type numérique, il n'y aura pas de problème car le saut de ligne sera considéré comme un délimiteur. Par contre, si cette prochaine lecture porte sur une variable de type caractère, elle va directement recevoir le caractère « saut de ligne » qui avait été mis à disposition lors de la lecture précédente.

2nd cas : l'utilisateur peut taper la réponse suivante : a« espace »b, puis valider par la touche Entrée. Dans ce cas, C1 va recevoir 'a', C2 v recevoir l'espace qui était sensé servir de délimiteur et qui est finalement considéré comme un caractère à part entière. Par conséquent, le 'b' et le « saut de ligne » resteront disponibles pour une prochaine lecture.

Il est possible dans certains cas d'esquiver ce genre de problème en prévoyant l'introduction d'un séparateur entre deux caractères à lire. Pour cela, il suffit d'introduire un espace entre les codes de format.

Exemple : scanf (« %d %c », &N, &C) ;

Dans ce cas, l'utilisateur devra taper une valeur numérique, un séparateur, un caractère, puis valider.

5/ Structures conditionnelle.

Pour conditionner l'exécution de certaines instructions sur un test, il existe en C l'équivalent du « si, alors, sinon, fin si » de l'algorithmie. Voici sa syntaxe :

```
if (<Condition>)
{
        bloc d'instructions 1
}
else
{
        bloc d'instructions 2
}
```

Exemple :

```
if (x < y)
{       printf (« %d »,x) ;
}
else
{       printf (« %d »,y) ;
}
```

Ce que l'on appelle « bloc d'instructions » est simplement un ensemble d'instructions contenues entre { et }. Comme en algorithmie, un bloc d'instructions peut contenir d'autres structure conditionnelles et donc d'autres bloc d'instructions. De plus, le bloc introduit par « else » peut tout-à-fait être ignoré.

La condition est l'équivalent du test de l'algorithmie. Elle peut porter aussi bien sur des valeurs numériques que des caractères, à condition de bien distinguer la signification des opérateurs de comparaison dans chaque cas. Voici les opérateurs que l'on peut utiliser, avec leurs différentes significations :

Opérateur	Signification pour les valeurs	Signification pour les caractères

	numériques	
==	égal	identique
<	strictement inférieur	de code inférieur
>	strictement supérieur	de code supérieur
<=	inférieur ou égal	de code inférieur ou égal
>=	supérieur ou égal	de code supérieur ou égal
!=	non égal	différent

Tous ces opérateurs de comparaison sont moins prioritaires que les opérateurs arithmétiques. Par conséquent, il est tout à fait possible de faire porter une condition sur des expressions.

Exemple :

```
if (x / 3 == y + 2)
{
        …
}
```

Remarque : Il peut arriver qu'une comparaison utilisant l'opérateur == sur des valeurs flottantes donne un résultat inattendu dû au fait que les flottants ne sont que des approximations des nombres réels.

5.1) Conditions composées.

Encore comme en algortihmie, il est possible d'utiliser des opérateurs logiques pour introduire plusieurs tests à l'intérieur d'une condition. Voici la notation de ces opérateurs :

Opérateur	Signification
&&	et
||	ou (inclusif)
!	non

« && » et « || » sont moins prioritaires que les opérateurs de comparaisons, alors que « ! » l'est plus.

6/ Structures de répétition : les boucles.

6.1) Boucles While et Do While..

L'équivalent en C de la boucle « Tant que » de l'algorithmie utilise la syntaxe suivante :

```
while (<Condition>)
```

{
<Bloc d'instructions>
}

L'équivalent en C de la boucle « Faire tant que » de l'algorithmie utilise la syntaxe suivante :

do
{
<Bloc d'instructions>
}
while (<Condition>) ;

Ces boucles s'utilisent en C exactement comme en algorithmie et comme dans la grande majorité des langages de programmation.

6.2) Boucle For.

L'équivalent en C de la boucle « Pour » de l'algorithmie utilise la syntaxe suivante :

for (<affectation de départ> ; <test d'arrivée> ; <instruction d'incrémentation>)
{
<Bloc d'instructions>
}

Attention, contrairement à l'algorithmie, la limite de la boucle for n'est pas une « valeur d'arrivée », mais un « test d'arrivée ».

Exemple de boucle « for » utilisant comme compteur la variable comp de type int :

```
for (comp = 0 ; comp < 10 ; comp ++)
    {
    printf(« %d », comp) ;
}
```

Exercices du cours

Ecrire un programme qui affiche le message « Bonjour ».
Ecrire un programme qui affiche une valeur entière contenue dans une variable.
Ecrire un programme qui affiche une valeur flottante sou s sa forme décimale et sous sa forme exponentielle.

Ecrire un programme qui lit un nombre (avec un dialogue explicite) et qui en affiche le double.
Ecrire un programme qui lit un caractère et qui l'affiche 3 fois de suite.
Ecrire un programme qui lit un nombre x et un caractère et qui affiche ce caractère x fois de suite.
Ecrire un programme qui lit 4 nombres et qui en affiche la moyenne.
Ecrire un programme qui lit deux prix et une quantité et qui en affiche le prix total.
Ecrire un programme qui lit 3 nombres et qui affiche le produit du plus grand par le plus petit.
Ecrire un programme qui lit un nombre et qui dit s'il est compris entre 100 et 200 ou pas.
Ecrire un programme qui lit un caractère et qui dit s'il est compris entre 'g' et 'p' ou pas.
Ecrire un programme qui lit un nombre x et qui affiche le message « Bonjour » x fois.
Ecrire un programme qui lit un nombre jusqu'à obtenir une valeur comprise entre 100 et 200.
Ecrire un programme qui lit un caractère jusqu'à obtenir une valeur comprise entre 'g' et 'p' ou entre 'G' et 'P'.
Ecrire un programme qui lit un nombre et qui le multiplie par 2 jusqu'à dépasser 10000 (afficher seulement le résultat final).
Même programme que précédemment, mais en affichant tous les résultats intermédiaires.
Ecrire un programme qui lit une phrase.
Ecrire un programme qui lit une phrase et qui affiche le nombre de caractères qu'elle contient.
Ecrire un programme qui lit une phrase et qui affiche le nombre de lettres qu'elle contient.
Ecrire un programme qui lit une phrase et qui affiche le nombre de mots qu'elle contient.
Ecrire un programme qui lit 2 nombres entiers x et n et qui affiche le résultat de xn.
Ecrire un programme qui lit un nombre x et qui affiche un triangle rectangle formé de x lignes de '*'.
Exemple : x = 5

```
*
**
***
****
*****
```

Ecrire un programme qui fait la lecture d'un nombre à virgule chiffre par chiffre.