LES FICHIERS

I. Organisation des fichiers

Il est difficile de trouver une notation algorithmique pour les fichiers. En effet, les fichiers représentent un concept de programmation très hétérogène, multiforme. Il y a des catégories, et dans les catégories, des sortes, et dans les sortes des espèces.

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

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

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

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

Le second type de fichier se définit a contrario : il rassemble les fichiers qui ne possèdent pas de structure de lignes (d'enregistrement). Les octets, quels qu'il soient, sont écrits à la queue leu leu. Ces fichiers sont appelés des **fichiers binaires**. Naturellement, leur structure différente implique un traitement différent par le programmeur. Tous les fichiers qui mémorisent pas de données structurées sont obligatoirement des fichiers binaires : cela concerne par exemple un fichier son, une image, un programme exécutable, etc. . Toutefois, on en dira quelques mots un peu plus loin, il est toujours possible d'opter pour une structure binaire même dans le cas où le fichier représente des données structurées.

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

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

II. Structure des enregistrements

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

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

Structure n°1

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

ou ainsi :

Structure n°2

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

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

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

L'avantage de la structure n°1 est son **faible encombrement en place mémoire** ; il n'y a aucun espace perdu, et un fichier texte codé de cette manière occupe le minimum de place possible. Mais elle possède en revanche un inconvénient majeur, qui est la **lenteur de la lecture**. En effet, chaque fois que l'on récupère une ligne dans le fichier, il faut alors parcourir un par un tous les caractères pour repérer chaque occurrence du caractère de séparation avant de pouvoir découper cette ligne en différents champs.

La structure n°2, à l'inverse, **gaspille de la place mémoire**, puisque le fichier est un vrai gruyère plein de trous. Mais d'un autre côté, la récupération des différents champs est très **rapide**. Lorsqu'on récupère une ligne, il suffit de la découper en différentes chaînes de longueur prédéfinie, et le tour est joué.

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

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

Types d'accès

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

On distingue :

- **L'accès séquentiel**: on ne peut accéder qu'à la donnée suivant celle qu'on vient de lire. On ne peut donc accéder à une information qu'en ayant au préalable examiné celle qui la précède. Dans le cas d'un fichier texte, cela signifie qu'on lit le fichier ligne par ligne (enregistrement par enregistrement).

- **L'accès direct** (ou **aléatoire**) : on peut accéder directement à l'enregistrement de son choix, en précisant le numéro de cet enregistrement. Mais cela veut souvent dire une gestion fastidieuse des déplacements dans le fichier.

- **L'accès indexé** : pour simplifier, il combine la rapidité de l'accès direct et la simplicité de l'accès séquentiel (en restant toutefois plus compliqué). Il est particulièrement adapté au traitement des gros fichiers.

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

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

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

Dans le cadre de ce cours, on se limitera volontairement au type de base : le **fichier texte en accès séquentiel.** Pour des informations plus complètes sur la gestion des fichiers binaires et des autres types d'accès, il vous faudra étudier la documentation du langage spécifique dans lequel vous voulez coder.

III. Instructions (fichiers texte en accès séquentiel)

Si l'on veut travailler sur un fichier, la première chose à faire est de l'**ouvrir**. En VB, cela se fait en attribuant au fichier un numéro de canal (ou à un nom logique dans d'autres langages). On ne peut ouvrir qu'un seul fichier par canal, mais quel que soit le langage, on dispose toujours de plusieurs canaux, donc pas de soucis.

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

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

Ces limitations sont très fortes. Il n'y a même pas d'instructions qui permettent de supprimer un enregistrement d'un fichier ! Toutefois, avec un peu d'habitude, on se rend compte que malgré tout, même si c'est fastidieux, on peut quand même faire tout ce qu'on veut avec ces fichiers séquentiels.

Pour ouvrir un fichier texte, on écrira par exemple :

```
// syntaxe VB : déclaration inutile
Ouvrir "Exemple.txt" sur 4 en Lecture
```

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

```
// syntaxe C
Var
monfichier : fichier séquentiel
…
Ouvrir "Exemple.txt" dans "monfichier" en Lecture
```

On utilise plutôt cette notation en algo (au BTS). Monfichier est le nom logique du fichier dans le programme et Exemple.txt est le nom "physique" utilisé par le système d'exploitation.

Manipulation du fichier

```
Var
Truc: chaine
Nom, Mail : chaines de 20 caractères
Tel : chaine de 10 caractères
Prénom : chaine de 15 caractères
```

//syntaxe VB

```
...
LireFichier 4, Truc
Nom ? Mid(Truc, 1, 20)
Prénom ? Mid(Truc, 21, 15)
Tel ? Mid(Truc, 36, 10)
Mail ? Mid(Truc, 46, 20)
```

//syntaxe algo/C

```
Lire (monfichier, Truc)
Nom ? sschaine(Truc, 1, 20)
Prénom ? sschaine(Truc, 1, 15)
Tel ? sschaine(Truc, 1, 10)
Mail ? sschaine(Truc, 1, 20)
```

L'instruction LireFichier (ou plus simlement Lire) récupère donc dans la variable spécifiée l'enregistrement suivant dans le fichier ("suivant" est compris par rapport au dernier enregistrement

lu). C'est en cela que le fichier est dit séquentiel. En l'occurrence, on récupère donc la première ligne, donc, le premier enregistrement du fichier, dans la variable Truc. Ensuite, le fichier étant organisé sous forme de champs de largeur fixe, il suffit de tronçonner cette variable Truc en autant de morceaux qu'il y a de champs dans l'enregistrement, et d'envoyer ces tronçons dans différentes variables. (C'est l'extraction de sous-chaines).

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

```
Variable Truc : chaine

Ouvrir "Exemple.txt" sur 5 en Lecture
Tantque Non EOF(5)        // ou Tantque Non EOF(monfichier)
     LireFichier 5, Truc
     …
FinTantQue
```

Et neuf fois sur dix également, si l'on veut stocker au fur et à mesure en mémoire vive les informations lues dans le fichier, on a recours à des tableaux. Et comme on ne savait pas d'avance combien il y aurait d'enregistrements dans le fichier, on ne sait pas davantage combien il doit y avoir d'emplacements dans les tableaux. Qu'importe, les programmeurs avertis que vous êtes connaissent la combine des tableaux dynamiques.

En rassemblant l'ensemble des connaissances acquises, nous pouvons donc écrire le prototype de lecture intégrale d'un fichier séquentiel, lecture qui recopie l'ensemble des informations en mémoire vive :

```
Var
Nom, Prénom, Tel, Mail : Tableaux[1..100] de Chaines

Ouvrir "Exemple.txt" sur 5 en Lecture
i ? -1
Tantque Non EOF(5)
     LireFichier 5, Truc
     i ? i + 1
     Nom[i] ? Mid(Truc, 1, 20)
     Prénom[i] ? Mid(Truc, 21, 15)
     Tel[i] ? Mid(Truc, 36, 10)
     Mail[i] ? Mid(Truc, 46, 20)
FinTantQue
```

```
Ouvrir "Exemple.txt" dans "monfichier" en Lecture

Tantque Non EOF(monfichier)
       LireFichier 5, Truc
       i ? i + 1
       Nom ? sschaine(Truc, 1, 20)
       Prénom  …
       …
FinTantQue
```

Ici, on a fait le choix de recopier le fichier dans quatre tableaux distincts. Comme on va le voir bientôt, il y a encore mieux, mais ne nous impatientons pas, chaque chose en son temps.

Pour une opération d'écriture, ou d'ajout, il faut d'abord constituer une chaîne équivalente à la nouvelle ligne du fichier. Cette chaîne doit donc être « calibrée » de la bonne manière, avec les différents champs qui « tombent » aux emplacements corrects. Le moyen le plus simple pour s'épargner de longs traitements est de procéder avec des chaînes correctement dimensionnées dès leur déclaration (la plupart des langages offrent cette possibilité) :

```
Ouvrir "Exemple.txt" sur 3 en Ajout
Var
Truc : chaine
Nom, Mail : chaines de 20 caractères
Prénom : chaine de 15 caractères
Tel chaine de 10 caractères
```

Une telle déclaration assure que quel que soit le contenu de la variable Nom, par exemple, celle-ci comptera toujours 20 caractères. Si son contenu est plus petit, alors un nombre correct d'espaces sera

automatiquement ajouté pour combler. Si on tente d'y entrer un contenu trop long, celui-ci sera automatiquement tronqué.

```
…
Nom ? "Jokers"
Prénom ? "Midnight"
Tel ? "0348946532"
Mail ? "allstars@rockandroll.com"
Truc ? Nom & Prénom & Tel & Mail
EcrireFichier 3, Truc
```

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

IV. Stratégies de traitement

Il existe globalement deux manières de traiter les fichiers textes :

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

Les avantages de la seconde technique sont nombreux, et 99 fois sur 100, c'est ainsi qu'il faudra procéder :

· la **rapidité** : les accès en mémoire vive sont des milliers de fois plus rapides (nanosecondes) que les accès aux mémoires de masse (millisecondes au mieux pour un disque dur). En basculant le fichier du départ dans un tableau, on minimise le nombre ultérieur d'accès disque, tous les traitements étant ensuite effectués en mémoire.
· la **facilité de programmation** : bien qu'il faille écrire les instructions de recopie du fichier dans le tableau, pour peu qu'on doive tripoter les informations dans tous les sens, c'est largement plus facile de faire cela avec un tableau qu'avec des fichiers.

Mais il vaut mieux en rester aux fichiers et ne pas passer par des tableaux quand on utilise de très gros fichiers. En effet, la recopie d'un très gros fichier en mémoire vive exige des ressources qui peuvent atteindre des dimensions considérables. Donc, dans le cas d'immenses fichiers (très rares, cependant, car pour les grosses bases de données, on utilise des SGBD), la recopie en mémoire peut s'avérer problématique.

V. Utilisation des données structurées

A. Données structurées simples

L'utilité d'une variable structurée (composée de plusieurs variables de type différent), lorsqu'on traite des fichiers texte (et même, des fichiers en général), saute aux yeux : car on va pouvoir calquer chacune des lignes du fichier en mémoire vive, et considérer que chaque enregistrement sera recopié dans une variable et une seule, qui lui sera adaptée. Ainsi, le problème du "découpage" de chaque enregistrement en différentes variables (le nom, le prénom, le numéro de téléphone, etc.) sera résolu

d'avance, puisqu'on aura une structure, un gabarit, en quelque sorte, tout prêt d'avance pour accueillir et prédécouper nos enregistrements.

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

Reprenons une fois de plus l'exemple du carnet d'adresses.

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

```
Structure Bottin
   Nom : chaine de 20 caractères
   Prénom : chaine de 15 caractères
   Tel chaine de 10 caractères
   Mail : chaine de 20 caractères
Fin Structure
```

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

```
Var
Individu : Bottin
```

Je pourrais remplir les différentes informations contenues au sein de la variable Individu de la manière suivante :

```
Individu ?“Joker“,“Midnight”,“0348946532”,“allstars@rock.com”
```

Mais cette notation est à éviter, car peu de langages l'acceptent. Il faut manipuler les variables structurées champs par champs.

```
Individu.Nom ? “Jokers“
Individu.Prénom ? “Midnight”
Individu.Tel ? “0348946532”
Individu.Mail ? “allstars@rockandroll.com”
```

Ainsi, écrire correctement une information dans le fichier est un jeu d'enfant, puisqu'on dispose d'une variable Individu au bon gabarit. Une fois remplis les différents champs de cette variable - ce qu'on vient de faire -, il n'y a plus qu'à envoyer celle-ci directement dans le fichier (ici, fichier de canal 3).

```
EcrireFichier 3, Individu
```

De la même manière, dans l'autre sens, l'opération de lecture dans le fichier Adresses sera considérablement simplifiée : la structure étant faite pour cela, je peux dorénavant me contenter de recopier une ligne du fichier dans une variable de type Bottin, et le tour sera joué. Pour charger l'individu suivant du fichier de canal 5 en mémoire vive, il me suffira donc d'écrire :

```
LireFichier 5, Individu
```

Et là, direct, j'ai bien mes quatre renseignements accessibles dans les quatre champs de la variable individu. Tout cela, évidemment, parce que la structure de ma variable Individu correspond parfaitement à la structure des enregistrements de mon fichier.

B. Tableaux de données structurées

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

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

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

```
Structure Bottin
   Nom : chaine de 20 caractères
   Prénom : chaine de 15 caractères
   Tel chaine de 10 caractères
   Mail : chaine de 20 caractères
Fin Structure
Mespotes : Tableau(1..100) de Bottin
Ouvrir “Exemple.txt” sur 3 en Ajout
i ? 0
Tantque Non EOF(5)
     i ? i + 1
     LireFichier 5, Mespotes(i)
FinTantQue
```

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

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

Récapitulatif général

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

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

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