*C*entre *I*nformatique pour les *L*ettres
et les *S*ciences *H*umaines

C++ : Leçon 8
Fichiers

Maintenant que nous commençons à disposer d'une portion significative de la puissance du langage C++, il serait agréable d'une part de pouvoir traiter d'autres données que celles fournies à l'aide du clavier pendant l'exécution du programme, et, d'autre part, d'être en mesure de conserver les résultats obtenus sous une forme un peu plus permanente qu'un simple affichage à l'écran. Ces deux problèmes peuvent être résolus en utilisant des fichiers.

1 - Qu'est-ce qu'un fichier ?

Les données manipulées par les programmes, ainsi que les résultats obtenus à l'issue de ces manipulations, sont représentés en mémoire (cf. Leçon 1). A l'heure actuelle, les caractéristiques des mémoires généralement utilisées imposent plusieurs restrictions inacceptables :
- L'information n'y est conservée que sous réserve d'une alimentation électrique permanente, et l'extinction de la machine conduit donc à la perte des résultats obtenus.
- Le dispositif électronique réalisant la fonction de mémoire est très coûteux, ce qui rend inenvisageable le stockage (même temporaire) de très gros volumes de données en mémoire.
- La mémoire n'est accessible que par l'ordinateur auquel elle appartient. La communication de données entre deux machines suppose donc que, à un moment donné, ces données soient représentées ailleurs qu'en mémoire.

Notion d'entrées/sorties

Pour contourner les limitations sus-citées, il est donc nécessaire de disposer de moyens permettant "l'exportation" de données, c'est à dire leur transfert de la mémoire de l'ordinateur vers un autre support.

D'un certain point de vue, le simple affichage à l'écran (que nous pratiquons depuis le premier TD), constitue déjà une exportation de données. Toutefois, pour ce qui est de lever les contraintes d'usage signalées ci-dessus, on ne peut malheureusement pas dire que l'affichage à l'écran constitue une piste très prometteuse...

Cette exportation peut être dirigée vers une grande variété d'appareils susceptibles de recevoir les données émises par l'ordinateur : disques durs, graveurs de CD, imprimantes et modems ne sont que les plus répandus des *périphériques de sortie* envisageables. L'intérêt de l'exportation de données est considérablement accru par la possibilité d'effectuer le transfert inverse : l'information peut également être importée en mémoire, à partir de divers *périphériques d'entrée*. Dans bien des cas, le même appareil peut assurer les deux types de transfert, méritant ainsi le nom de *périphérique d'entrée/sortie*.

Le plus banal des périphériques d'entrée est, bien entendu, le clavier. Il est sans doute inutile d'insister sur les graves défauts que présente ce dispositif lorsque le volume de données est très important...

Le stockage des données sur un support autre que la mémoire suppose l'adoption d'un certain nombre de conventions permettant l'accès à ce support et l'interprétation correcte des données qu'il contient. La prise en charge de ces détails techniques est assurée par le système d'exploitation, éventuellement assisté par la présence de *drivers*, c'est à dire de programmes spécialisés dans le pilotage de certains périphériques. Du point de vue de l'utilisateur, un grand nombre de périphériques adoptent la notion de fichier, ce qui permet d'attribuer des noms significatifs aux diverses copies de sections de mémoire qui figurent sur un même support, et de les manipuler de façon unifiée (c'est à dire sans avoir à tenir compte de la nature du périphérique), à l'aide des interfaces utilisateur du système d'exploitation et de la plupart des applications. Dans le contexte de ce cours, nous adopterons donc la définition suivante[1] :

Un fichier est une représentation du contenu d'un fragment de la mémoire, stockée sur un périphérique qui permet au système d'exploitation d'utiliser un nom pour désigner cette représentation.

Outre son nom, un fichier possède un certain nombre de caractéristiques (taille, date de création, etc.) qui peuvent éventuellement être stockées par le système d'exploitation et communiquées à l'utilisateur.

[1] D'autres définitions sont parfois adoptées, notamment parce que, du point de vue du programme qui les reçoit, il n'y a pas de réelles différences entre des données provenant d'un fichier (au sens ou nous l'entendons ici) et des données provenant d'une autre source. Il me semble toutefois préférable de restreindre l'usage du mot "fichier" (*file*) et d'employer le terme plus générique de "flux de données" (*data stream*) lorsque cette restriction n'est pas de mise.

Un texte affiché à l'écran ou imprimé sur une feuille de papier ne constitue donc pas un fichier, non plus que les flux de données en provenance du clavier, d'un modem ou d'un scanner (même si, bien entendu, les informations provenant de ces périphériques peuvent, comme toute information présente en mémoire, être par la suite copiées dans un fichier).

Pour nos programmes, l'utilisation de fichiers de données va donc se traduire par deux possibilités nouvelles : le traitement de données qui n'auront pas à être saisies au clavier pendant l'exécution du programme (elles pourront être lues dans un fichier) et la sauvegarde des résultats en vue d'une exploitation ultérieure (ils pourront être enregistrés dans un fichier).

Fichiers bruts et fichiers formatés

Puisqu'un fichier contient "une représentation du contenu d'un fragment de la mémoire", l'hypothèse la plus naturelle est sans doute de penser qu'il s'agit d'une copie, octet par octet, des états des cases mémoire correspondant au fragment en question. Si cette façon de procéder est effectivement souvent employée, ce n'est toutefois pas la seule qui soit utilisable. Imaginons, par exemple qu'il s'agisse d'effectuer une sauvegarde d'une zone de mémoire dont l'état électrique serait le suivant :

Selon le procédé que nous venons d'évoquer, l'information stockée dans le fichier serait le strict équivalent binaire de cette zone de mémoire, soit les quatre octets

0101 0000
0100 0101
0101 0010
0100 1001

Il s'agirait alors d'un fichier contenant ce qu'on appelle des données "brutes" (*raw data*, en anglais) ou, parfois, une "image mémoire". L'inconvénient des données brutes est que le contenu du fichier est difficile à interpréter (aussi difficile à interpréter, en fait, que l'état de la mémoire auquel il correspond...). Face aux quatre octets de notre exemple, de nombreuses hypothèses sont envisageables : s'agit-il de caractères codés en ASCII ? Ce seraient alors les lettres P,E,R et I. Mais il peut tout aussi bien s'agir d'un nombre et, dans ce cas, est-il codé comme un entier ou comme un décimal ? Et ce fichier a-t-il été créé sur un système *little endian* ou *big endian* ? S'il s'agit d'une machine à processeur Intel (*little endian*), la valeur entière est 1 230 128 464 et la valeur décimale est 861 269, mais s'il s'agit d'une machine à processeur Motorola (*big endian*), l'entier est 1 346 720 329 et le décimal est 13 242 016 768. Sans compter qu'il pourrait bien s'agir de deux valeurs codées chacune sur deux octets... Ou de deux valeurs codées sur un octet, suivies d'une valeur codée sur deux octets... En pratique, il est préférable de n'employer des données brutes que lorsque le fichier est destiné à être relus par le programme qui l'a créé, ou par un second programme étroitement associé au premier. Dans ce cas, il suffit de veiller à ce que les fonctions respectivement responsables de l'écriture et de la lecture du fichier soient cohérentes, et aucun problème d'interprétation ne se pose[2].

Lorsqu'un fichier est destiné à être relu par un programme étranger à celui qui l'a créé, il devient nécessaire d'adopter un *format*, c'est à dire des conventions qui permettront au lecteur de retrouver dans les données la signification qu'y a placé l'auteur. Il existe de nombreux formats de fichiers destinés à permettre l'échange des divers types de données couramment manipulées par les ordinateurs : images (Gif, JPEG...), sons (Wave, MP3...), texte mis en forme (RTF, PDF...), etc. Le plus simple de tous ces formats, et sans doute celui dont l'usage est le plus général, est celui qu'il est convenu d'appeler le format *texte*.

Le principe d'un fichier texte est très simple : chaque octet doit en être interprété comme étant le code ASCII d'un caractère. La pratique souffre malheureusement de l'existence de plusieurs versions du format texte, liées non seulement au fait que le code ASCII ne définit que les caractères nécessaires aux américains (et qu'il existe plusieurs extensions différentes de ce code permettant de représenter les minuscules accentuées), mais aussi au fait qu'un désaccord existe quant à la façon dont il convient de représenter le passage à la ligne : pour certains un

[2] Il reste quand même la question des changements de système : les fichier bruts créés par un programme exécuté sous Windows, par exemple, ne pourront pas être exploités normalement par le même programme, recompilé et exécuté sous Mac OS (ne serait-ce qu'à cause du problème little endian / big endian...)

"saut de ligne" (le caractère *line feed*, noté LF, code ASCII 10 ou 0x0A) suffit, alors que, pour d'autres, il doit être précédé d'un "retour chariot" (le caractère *carriage return*, noté CR, code ASCII 13 ou 0x0D). En dépit de ce léger flou, le format texte constitue certainement le plus universel des formats de fichiers, en particulier (contrairement à ce que son nom suggère) pour les données numériques.

Reprenons notre exemple, et supposons que nous cherchions à représenter dans un fichier la quantité 1 230 128 464. Il nous suffit pour cela de représenter les 10 caractères correspondant à la représentation de ce nombre en base 10. Chacun de ces caractères est représenté par son code ASCII, ce qui nous donne :

0011 0001
0011 0010
0011 0011
0011 0000
0011 0001
0011 0010
0011 1000
0011 0100
0011 0110
0011 0100

L'inconvénient principal de cette méthode apparaît immédiatement : elle n'est guère économique, puisque nos *quatre* octets de mémoire sont représentés par *dix* octets dans le fichier. En revanche, si l'on sait qu'il s'agit d'un fichier texte, la valeur est représentée sans ambiguïté et peut être reconnue par tous les systèmes, qu'ils soient *little endian* ou *big endian*, et qu'ils préfèrent les entiers ou les décimaux.

> Dans le contexte d'un système d'exploitation Microsoft, le fait qu'un fichier est interprétable comme une suite de codes ASCII n'est signalé que par l'utilisation d'une extension ".txt" dans le nom du fichier. Si vous ne souhaitez pas délibérément créer des complications, utilisez systématiquement cette extension pour vos fichiers "texte", et ne l'utilisez que dans ce cas.

Bien entendu, si notre fichier comporte plusieurs valeurs, il convient d'indiquer l'endroit où l'une finit et l'autre commence. On insère donc des *caractères séparateurs* qui peuvent être, selon les besoins, des sauts de lignes, des tabulations ou de simples espaces. Beaucoup de logiciels interprètent des valeurs séparées par des tabulations comme appartenant aux différentes colonnes d'une même ligne, alors que les passages à la ligne indiquent, assez logiquement, une nouvelle ligne. Un fichier texte organisé de cette façon pourra être directement relu par un tableur ou un traitement de texte, par exemple, ce qui permet d'effectuer des traitements complémentaires, ou, plus simplement, de réaliser une mise en page raffinée en vue d'une impression.

2 - Manipulation de fichiers en C++ : principes généraux

Pour des raisons techniques, les mécanismes de manipulation de flux ne font pas partie du langage C++ proprement dit, mais sont déportés dans la librairie standard[3]. Cette déportation implique la nécessité d'inclure explicitement les déclarations nécessaires. Concrètement :

Si une fonction manipule un fichier de données, le fichier .cpp dans lequel elle figure doit comporter dans son en-tête la directive
```
#include "fstream.h"
```

La technique utilisée pour opérer sur un fichier de données est indirecte : elle repose sur la création d'une variable d'un type spécial, que l'on associe au fichier que l'on souhaite utiliser. Une fois cette association réalisée, le programme n'agira plus que sur la variable qui, de son point de vue, *représente* désormais le fichier.

> La répercussion sur le fichier des actions que le programme semble effectuer sur la variable qui représente ce fichier est effectuée automatiquement : toute la "magie" réside dans le type de variable utilisé, qui n'a précisément été conçu que pour cela. Cet automatisme ne doit cependant pas vous conduire à confondre le fichier proprement dit (tel que vous pouvez le "voir" à l'aide du système d'exploitation, lorsque vous en faites une copie sur une disquette, par exemple) et la variable que votre programme lui a (temporairement) associé.

[3] Ceci ne compromet pas l'universalité de ces mécanismes, car la librairie standard est aussi strictement normalisée que le langage lui-même.

Deux types de variables peuvent être utilisés pour représenter les fichiers :
- la classe `ofstream` (output file stream) permet d'insérer (put) des données dans le fichier associé à la variable ou, en d'autres termes, d'écrire dans le fichier ;
- la classe `ifstream` (input file stream) permet d'extraire (get) des données du fichier associé à la variable ou, en d'autres termes, de lire le fichier ;

Ces classes sont pourvues de fonctions membre qui vont nous permettre d'opérer sur les instances et donc, indirectement, sur les fichiers qui leur sont associés. Elles comportent également des variables membre qui nous restent invisibles, mais qui sont modifiées lors des opérations que nous effectuons, y compris dans le cas d'une simple lecture de données. En conséquence :

Lorsqu'une fonction manipule un fichier à l'aide d'une variable associée qui lui est transmise par passage de paramètre, ce paramètre doit être un pointeur (sur `ofstream` ou `ifstream`, selon le cas).

> En effet, si la fonction appelée opère sur une simple copie de la variable associée au fichier, les opérations de lecture ou d'écriture qu'elle effectuera n'auront aucune conséquence sur l'état des variables membre de la variable originale[4], qui ne reflètera donc plus l'état réel du fichier. Si cette variable "désynchronisée" est ensuite utilisée pour de nouvelles opérations de lecture ou d'écriture, le résultat obtenu aura peu de chances d'être satisfaisant...

Remarquons, au passage, que la déclaration d'une telle fonction mentionne nécessairement le type de la variable associée au fichier utilisé, puisqu'il s'agit là d'un des paramètres de la fonction. Il est donc nécessaire que la directive `#include "fstream.h"` figure en tête du fichier .h dans lequel la fonction est déclarée (ce qui assure, indirectement, la présence de cette directive dans tous les fichiers .cpp qui incluent ce fichier .h, et y rend donc optionnelle son insertion explicite).

En l'absence de spécification contraire, l'exploitation d'un fichier se fait de façon séquentielle, à partir du début du fichier, et le déplacement est implicite.

> En d'autres termes, si un programme écrit deux fois de suite dans un fichier, les deux informations s'y succèdent dans l'ordre chronologique des écritures. Inversement si un programme lit deux fois de suite dans un fichier, il obtient deux informations qui se suivent dans le fichier, et non pas deux fois la même information.

3 - Ouverture d'un fichier

Toute manipulation de fichier repose donc sur l'usage d'une variable : de type `ofstream` s'il s'agit d'insérer des données dans le fichier, ou de type `ifstream` s'il s'agit d'en extraire. Cette variable doit, en outre, être associée au fichier visé. Lorsqu'on procède à cette association, on dit qu'on ouvre le fichier, ce qui signifie que celui-ci est ensuite prêt, selon le cas, à recevoir les données que l'on souhaite y placer ou à fournir les données que l'on souhaite lire.

> Le fait que le fichier soit "prêt à..." n'implique évidemment aucun phénomène visible pour l'utilisateur du programme. Si celui-ci doit disposer d'une fenêtre lui permettant de consulter ou de modifier le contenu du fichier, c'est à votre programme de prendre en charge la création de la fenêtre en question et la gestion de son contenu, ainsi que les actions de l'utilisateur.

Initialisation de la variable associée

La méthode la plus simple pour ouvrir un fichier est d'initialiser la variable qui doit lui être associée à l'aide d'une chaîne de caractères décrivant le chemin d'accès au fichier.

```
1  ifstream lesDonnees("c:\\Mes documents\\data.txt");
2  ofstream mesResultats("c:\\Mes documents\\resultats.txt");
```

> Le format de la chaîne de caractères dépend du système d'exploitation utilisé. Dans le cas d'un système Microsoft, la segmentation dossier, sous dossier, fichier est indiquée au moyen d'une barre oblique inverse. Il se trouve que ce caractère possède un sens particulier lorsqu'il figure dans une chaîne littérale en C++. Il s'agit, en effet, du "caractère d'échappement", qui indique que le caractère suivant ne doit pas être interprété littéralement, mais possède un sens spécial. Ainsi, par exemple, la présence de \n dans la définition littérale d'une chaîne n'insère pas dans la chaîne une barre oblique inverse suivie d'une lettre n minuscule, mais

[4] Si ceci ne vous semble pas TOTALEMENT évident, revoyez d'urgence les Leçons 5 et 6.

un caractère de passage à la ligne. L'insertion d'une barre oblique inverse dans une chaîne littérale nécessite le redoublement de ce caractère dans le texte source.

Ce problème ne se pose pas sur un Macintosh, où la segmentation des chemins d'accès est assurée par le caractère ":", qui n'a aucune signification particulière lorsqu'il apparaît dans une chaîne littérale C++. Le fragment de code précédent devient alors :

```
1  ifstream lesDonnes("Macintosh:Mes documents:data.txt");
2  ofstream mesResultats("Macintosh:Mes documents:résultats.txt");
```

Cette façon d'ouvrir les fichiers présente plusieurs caractéristiques notables :
- Si le chemin d'accès comporte un dossier qui n'existe pas, l'ouverture échoue.
- Si le chemin est correct, mais que le fichier n'existe pas, il est créé[5].
- Si le fichier existe, il est ouvert. S'il s'agit d'une ouverture en vue d'une insertion de données, la longueur du fichier est ramenée à 0 (ce qui signifie que son contenu antérieur est perdu).

Si ces caractéristiques ne conviennent pas à l'usage envisagé pour le fichier, il faut préciser plus finement l'effet souhaité, en fournissant une seconde valeur lors de l'initialisation de la variable associée au fichier. Cette valeur, qui indique le mode d'ouverture désiré, peut être fixée à l'aide des constantes suivantes :

`ios::app`	Si le fichier existe, son contenu est conservé et toutes les données insérées seront ajoutées en fin de fichier (app est l'abréviation de *append*).
`ios::nocreate`	Si le fichier n'existe pas, il ne sera pas créé, mais il y aura échec de l'ouverture.
`ios::noreplace`	Si le fichier existe déjà il y aura erreur d'ouverture (sauf si le mode `ios::app` est également spécifié).
`ios::binary`	Si ce mode n'est pas spécifié, le fichier est ouvert en mode "texte", ce qui signifie que divers ajustements seront automatiquement effectués, en fonction des nécessités imposées par le système d'exploitation. Ces ajustements concernent notamment la représentation du passage à la ligne, de façon à ce que l'insertion du caractère '\n' se traduise toujours par la présence dans le fichier du (ou des) caractères que le système d'exploitation considère être la bonne représentation d'un saut de ligne[6]. Attention à ne pas confondre le mode texte avec le format texte. Le mode texte *peut* être appliqué aussi bien à des fichiers au format texte qu'à d'autres types de fichiers. Notez cependant que les ajustements réalisés en mode texte ont *peu de chances de donner des résultats intéressants* si le fichier concerné *n'est pas* au format texte !

Il est possible de combiner plusieurs de ces valeurs à l'aide de l'opérateur "OU bitaire"[7] noté `|`. Ainsi, si l'on souhaite compléter un fichier qui existe déjà, on peut écrire :

```
ofstream monFichier("C:\\essai.txt", ios::nocreate | ios::app);
```

Le mode d'ouverture spécifié dans cet exemple nous garantit à la fois que nous ouvrons un fichier déjà présent (à cause de `ios::nocreate`) et que nous conservons son contenu antérieur (grâce à `ios::app`).

La fonction `open()`

Si l'initialisation d'une variable est une opération particulièrement simple, elle présente en revanche une limitation évidente : elle ne peut avoir lieu qu'une seule fois. Il arrive pourtant qu'une même variable intervienne dans plusieurs ouvertures (ou tentatives d'ouvertures...) successives. Les classe `ofstream` et `ifstream` proposent, pour répondre à ce besoin, une fonction membre nommée `open()` qui peut accepter comme paramètre un chemin d'accès, comme dans l'exemple suivant :

---

[5] Notez donc bien que l'ouverture d'un fichier ne conduira jamais à la création d'un dossier ou sous dossier. Ce type d'opérations nécessite l'appel d'une fonction propre au système d'exploitation utilisé, fonction qui est normalement fournie dans une librairie accompagnant le compilateur.

[6] Ainsi, dans un fichier ouvert en mode texte sous Windows, par exemple, l'insertion du caractère '\n' donne lieu à l'écriture de DEUX octets dans le fichier : \r et \n.

[7] L'usage des opérateurs bitaires est présenté en détails dans l'Annexe 5, qui vous permettra de comprendre pourquoi la conjonction de deux modes d'ouverture est obtenue à l'aide d'un opérateur OU et non, comme on pourrait s'y attendre, à l'aide d'un ET.

```
ofstream monFichier;
monFichier.open("c:\\essai.txt"); //ouverture "normale"
```

La fonction `open()` peut aussi recevoir une spécification de mode d'ouverture :

```
ofstream fichierACompleter;
fichierACompleter.open("c:\\ilGrandit.txt",ios::app);
```

Qu'elle s'effectue par initialisation ou par appel explicite de la fonction `open()`, l'ouverture d'un fichier est une opération dont le succès ne peut être garanti. Il faut donc se garder de supposer qu'elle s'est déroulée sans encombre, et aucune opération d'écriture dans un fichier ne devrait jamais être entreprise sans avoir tout d'abord vérifié que l'ouverture de celui-ci n'a pas échoué.

Les échecs d'ouverture

On détecte l'échec d'une ouverture de fichier en testant directement la valeur de la variable que l'on vient de tenter d'associer au fichier. Ceci est illustré dans l'exemple suivant, qui suppose qu'il existe une fonction `demandeLeChemin()`, dont le rôle est de renvoyer un chemin d'accès obtenu en posant la question à l'utilisateur :

```
ofstream monFichier;
do
   {
   monFichier.open(demandeLeChemin());
   } while (! monFichier);
```

La cause la plus probable d'un échec d'ouverture est en effet une erreur de chemin. La réaction la plus naturelle est donc de redemander ce chemin à l'utilisateur... en espérant qu'il comprendra quelle est son erreur !

> Dans un environnement graphique, les chemins sont normalement obtenus par l'intermédiaire d'un dialogue qui permet à l'utilisateur du programme d'explorer l'arborescence des fichiers au lieu d'avoir à taper l'intégralité du chemin au clavier. L'apparition de chemins invalides devient donc assez peu probable. Ceci ne dispense évidemment pas de vérifier que l'ouverture n'a pas échoué, mais on peut envisager de faire l'économie d'une réponse très élaborée en cas d'échec.

La conduite à tenir en cas d'échec lors de l'ouverture d'un fichier dépend largement de la nature du programme qui essuie cet échec. Dans les exemples qui vont suivre, nous nous bornerons à n'opérer sur le fichier que dans le cas où l'ouverture a réussi, le cas contraire ne pouvant être traité de façon vraisemblable en l'absence de contexte.

4 - Fichiers au format texte

Nous avons souligné l'intérêt que présente le format "texte" lorsque les fichiers doivent pouvoir être ouverts à l'aide de différents logiciels. En C++, l'utilisation de ce type de fichier est rendue facile par l'usage des opérateurs d'insertion et d'extraction.

Le format d'un fichier n'est pas déterminé par la façon dont il est ouvert, mais par son contenu. Si un fichier est écrit à l'aide de l'opérateur d'insertion, il est au format texte.

Les sorties formatées

Une sortie formatée est obtenue à l'aide de l'opérateur d'insertion dans un flux, noté `<<`. Cet opérateur présente un avantage énorme : il assure une représentation correcte pour tous les types de données qu'il accepte !

> Les types standard sont, bien entendu, tous reconnus par l'opérateur d'insertion. Pour ce qui est des classes, il est relativement facile de les rendre compatibles avec l'usage de cet opérateur, et une classe pour laquelle cette opération a un sens devrait toujours permettre à ses instances de se décrire elles-mêmes de cette façon lorsqu'on le leur demande. La manière de rendre une classe compatible avec l'opérateur d'insertion sera étudiée dans la Leçon 15.

Le fragment de code suivant définit et initialise trois variables de types prédéfinis, puis définit une variable `ofstream` qu'il utilise ensuite pour insérer dans un fichier une représentation du contenu des trois premières variables.

```
int n = 4;
double x = 3.7;
char c = 'y';
ofstream monFichier("essai.txt");
if (monFichier)
   {//l'ouverture a réussi, on peut donc écrire
   monFichier << n;
   monFichier << x;
   monFichier << c;
   }
```

Si, après exécution de ce fragment de code, on ouvre le fichier essai.txt avec un éditeur de texte, son contenu apparaît sous la forme suivante :

```
43.7y
```

La présence de caractères de séparation faciliterait l'interprétation correcte de ces données. L'insertion d'une tabulation est obtenue à l'aide du caractère '\t', et celle d'un saut de ligne à l'aide du caractère '\n'. Une alternative pour l'insertion d'un saut de ligne est proposée sous la forme du manipulateur `endl`.

> Etant donné que nous n'utiliserons pas les flux pour effectuer des formatages complexes de textes, la description systématique des manipulateurs et de leur usage peut être reportée à un futur indéterminé. Il n'y a pas vraiment d'inconvénient à considérer `endl` comme une simple écriture alternative de '\n' dans le contexte de l'opérateur d'insertion.

Si l'on ajoute que plusieurs opérateurs d'insertions peuvent être chaînés les uns à la suite des autres dans une même instruction, les trois dernières lignes de l'exemple précédent peuvent être avantageusement remplacées par :

```
monFichier << n << '\t' << x << endl;
monFichier << c << '\n';
```

L'exécution de cette nouvelle version du code crée un fichier qui apparaît ainsi lorsqu'il est ouvert avec un éditeur de texte (remarquez la ligne vide finale (3) créée par l'insertion de '\n') :

```
4       3.7
y

```

Les entrées formatées

L'opérateur d'extraction d'un flux, noté >>, permet d'obtenir une entrée formatée. Il est bien entendu nécessaire que la séquence de caractères extraite puisse être interprétée comme la représentation d'une valeur compatible avec le type de la variable destinée à la recevoir.

> Lors d'une entrée formatée concernant des données numériques, il ne s'agit bien entendu pas simplement de recopier les octets présents dans le fichier dans les cases mémoire destinées à stocker les données ! Un calcul doit être effectué sur les caractères extraits du fichier, de façon à déterminer quelle valeur ils représentent. Cette valeur doit ensuite être représentée conformément aux conventions correspondant au type attribué à la zone de mémoire où elle va être stockée.

Le fichier créé au paragraphe précédent peut être relu par le code suivant :

```
int entier;
double decimal;
char caractere;
ifstream leFichier = "essai.txt";
if(leFichier)
   {//l'ouverture a réussi, on peut donc lire
   leFichier >> entier;
   leFichier >> decimal;
   leFichier >> caractere;
   }
```

L'opérateur d'extraction reconnaît les passages à la ligne et les tabulations comme des caractères de séparation, ce qui est assez naturel. Ce qui est plus inattendu, c'est que :

Les espaces constituent des séparateurs pour l'opérateur d'extraction. Celui-ci n'est donc pas capable d'extraire en une seule fois un texte comportant plusieurs mots.

Il est évidemment possible d'extraire le texte mot par mot, mais cela n'est guère pratique. La classe `ifstream` possède une fonction membre qui permet de contourner cette difficulté.

La fonction `getline()`

Cette fonction utilise trois paramètres :
- Le premier est un "pointeur sur `char`" qui doit contenir l'adresse de la zone de mémoire qui va accueillir les caractères extraits du fichier.
- Le second est le nombre maximum de caractères qui doivent être extraits du fichier. Ce paramètre permet de garantir qu'un texte d'une longueur inattendue ne créera pas de problèmes en débordant de la zone de mémoire prévue à son intention.
- Le troisième paramètre est un caractère qui indique quel est le séparateur matérialisant la fin du texte à extraire. Ce paramètre est doté d'une valeur par défaut : le retour à la ligne '\n'. Si l'on transmet à `getline()` une autre valeur pour ce troisième paramètre, il est possible d'extraire en une seule opération un texte comportant plusieurs paragraphes.

> Un cas particulier intéressant : lorsqu'on lit un fichier texte, utiliser EOF (le code de fin de fichier texte, alias **E**nd **O**f **F**ile) comme délimiteur final permet à `getline()` d'extraire en une seule fois tout le texte, jusqu'à la fin du fichier (à condition, bien entendu, que la zone de mémoire accueillant les données soit d'une taille suffisante).

L'opérateur d'extraction et la fonction `getline()` n'ont, malheureusement, pas la même façon de gérer les caractères séparateurs, ce qui conduit à des complications lorsqu'un appel à `getline()` intervient immédiatement après une extraction ayant utilisé l'opérateur `>>`. Le problème est que cet opérateur n'extrait que la donnée précédant le prochain séparateur, ce dernier étant donc considéré comme le premier caractère non encore extrait. Si l'opération de lecture suivante est effectuée à l'aide d'un appel à `getline()` utilisant ce séparateur, aucune donnée ne sera donc extraite. Il existe plusieurs façon de résoudre ce problème. On peut, par exemple, éliminer explicitement le séparateur en faisant appel à la fonction `ignore()`. Cette fonction extrait (sans rien en faire de particulier) un nombre maximum de caractères spécifié par son premier paramètre, jusqu'à ce qu'elle rencontre le caractère spécifié par son second paramètre. Si `leFichier` comporte une valeur entière et une phrase de moins de 100 caractères séparées l'une de l'autre par un saut de ligne, on peut affecter la valeur entière à `uneVar` et copier la phrase à l'adresse contenue dans `unPointeur` en écrivant :

```
1  leFichier >> uneVar;                 //extraction de la valeur entière
2  leFichier.ignore(1,'\n');            //élimination "manuelle" du saut de ligne
3  leFichier.getline(unPointeur,100);   //lecture de la phrase
```

Un programme qui lit un fichier de données en utilisant tantôt l'opérateur d'extraction, tantôt la fonction `getline()` demande une attention particulière lors de sa mise au point.

5 - Fermer un fichier

Lorsqu'une variable associée à un fichier cesse d'exister (à la fin du bloc où elle est définie, comme toutes les variables), le fichier est automatiquement refermé. Il n'est donc pas nécessaire de fermer explicitement un fichier à la fin d'une fonction qui l'a ouvert à l'aide d'une de ses variables locales. Il arrive toutefois qu'une fermeture explicite s'avère nécessaire. C'est en particulier le cas si l'on souhaite associer un nouveau fichier à la même variable. On fait alors appel à la fonction membre nommée `close()`, comme dans l'exemple suivant :

```
1  ofstream monFichier("essai_1.txt");
2  monFichier << "Petite démonstration";
3  monFichier.close();     //fermeture du fichier essai_1.txt
4  monFichier.open("essai_2.txt");
5  monFichier << "On est dans le second fichier";
```

> Comme le montre l'exemple ci-dessus, un fichier peut être refermé à l'aide de `close()`, même s'il n'a pas été ouvert par `open()` mais par initialisation de la variable associée.

6 - Bon, c'est gentil tout ça, mais ça fait déjà 8 pages. Qu'est-ce que je dois vraiment en retenir ?

1) Un programme qui utilise des fichiers de données doit inclure fstream.h
2) Pour écrire dans un fichier, on déclare une variable de type `ofstream` qu'on initialise avec le chemin d'accès du fichier. On vérifie d'abord que l'ouverture s'est bien passée, puis on utilise l'opérateur d'insertion (<<) pour envoyer dans le fichier les données que l'on veut y voir figurer.
3) Pour relire le fichier ainsi créé, on déclare une variable de type `ifstream` qu'on initialise avec le chemin d'accès du fichier. On vérifie d'abord que l'ouverture s'est bien passée, puis on utilise l'opérateur d'extraction (>>) pour envoyer dans des variables appropriées les informations extraites du fichier.
4) Pour vérifier qu'une ouverture s'est déroulée normalement, il suffit de tester directement la variable de type `ofstream` ou `ifstream` :

```
if(monFichier)
  //tout va bien
else
  //on arrête les frais
```

5) Lorsqu'on doit passer un fichier comme paramètre à une fonction, il faut passer un pointeur sur la variable associée.
6) Dans les cas où les cinq points précédents ne suffisent pas à résoudre le problème, on se reporte à la Leçon 8, qui explique la différence entre fichiers texte et fichiers bruts et indique comment mêler les usages de `<<` et de `getline()`. Dans les cas encore plus épineux, on se reporte à l'Annexe 2, qui explique comment utiliser des entrées et sorties brutes, comment détecter et interpréter les différents cas d'erreur, comment contrôler la position à l'intérieur du fichier où auront lieu les lectures ou les écritures et comment utiliser un même fichier à la fois en lecture et en écriture.

7 - J'ai rien compris, est-ce que quelqu'un d'autre pourrait m'expliquer ça un peu plus clairement ?

De nombreux ouvrages consacrés au langage C++ traitent assez peu (voir pas du tout) de la manipulation des flux en général, et des fichiers en particulier. Un des prétextes de cette lacune est que la manipulation des flux ne fait pas partie du langage proprement dit, mais seulement de la librairie standard. Un autre est que certains programmeurs préfèrent encore s'appuyer sur la bibliothèque d'entrées/sorties du langage C (qui reste disponible en C++).

Dans le cas de notre manuel, il semble que l'absence totale de traitement de la question des flux de données en C++ soit attribuable à une troisième cause : Horton a choisi de ne s'intéresser aux fichiers que dans le contexte du modèle document/vue associé aux applications basées documents créées à l'aide des MFC. L'intérêt évident de cette technique ne doit pas faire oublier qu'elle est dénuée de toute portabilité : si vous ne connaissez qu'elle, vous ne savez utiliser des fichiers que sous Microsoft Windows...

En dépit de certains défauts, les flux de la librairie standard restent le seul moyen vraiment portable de créer des fichiers prenant en compte l'existence des classes. Je ne connais malheureusement pas de présentation de leur usage qui mérite réellement d'être signalée.

8 - Pré-requis de la Leçon 9

Aucune des Leçons ultérieures n'exige réellement que vous maîtrisiez le contenu de la Leçon 8. Il est cependant rare qu'un projet présentant un certain intérêt puisse se dispenser d'utiliser des fichiers de données, et vous serez certainement conduit à consulter régulièrement la Leçon 8, et parfois même l'Annexe 2 : Techniques complémentaires pour la gestion des fichiers.