Entrées/Sorties - Fichiers

On regroupe sous le terme entrées/sorties toute la gestion des données **externes au programme**, contenues sur des supports externes à la mémoire du programme qui seront **consommées/reçues** ou **produites/envoyées de/vers** ces **supports externes**. Parmi ces supports on trouve bien sûr les disques internes, externes, disquettes, clefs USB, etc. mais aussi le clavier et l'écran.

La gestion des entrées/sorties **ne dépend donc pas** directement **du programme**, mais du système d'exploitation et n'est donc pas prise en charge par le langage C/C++ mais par des librairies externes. Le C fournit des librairies de gestion des entrées/sorties dont le fichier d'entêtes stdio.h, et le C++ fournit sa propre version dont les points d'entrées principaux se trouvent dans les fichiers d'entêtes iostream et fstream.

Les fonctions et opérateurs fournies permettent de mettre en relation les programmes et les données externes.

En C++ on utilise facilement un terme générique pour parler autant de fichiers que de périphériques : on parle de flux (parfois de flot en français). Ce terme exprime bien l'idée de flux de données dans lesquelles on ajoute ou extrait des informations.

Du point de vue du programme un flux est donc un producteur et/ou un consommateur de données.

Du point de vue du système d'exploitation, qui conserve et gère le flux on peut lui donner certaines caractéristiques en disant qu'il s'agit d'un ensemble d'informations :
- plus ou moins organisé, structuré
- plus ou moins lisible,
- formant un tout,
- repéré, identifié, nommé,
- enregistré sur un support externe en ce sens où sa durée de vie persiste après l'arrêt de la machine (~ clavier/écran).

En résumé dans une problématique de programmation, nous devons voir un fichier par le point de vue du programme tout en respectant (subissant) les contraintes du système d'exploitation.

Codage / décodage ?

Un fichier est une suite de bits, organisés généralement en octets et représentant des données, des informations ayant du sens pour le programmeur/l'utilisateur. Ces informations sont donc codées et une partie du problème est leur décodage.

Fichiers binaires

Le codage des données dans le fichier est le même que celui en mémoire.

Avantages :
- le travail est simple, il s'agit de copier des octets directement, sans se préoccuper du type des données.

Inconvénients :
- le contenu du fichier n'est pas directement lisible,
- le codage mémoire étant dépendant de l'architecture de la machine, la portabilité n'est pas assurée.

Fichiers textes

Dans un fichier texte les informations sont enregistrées en caractères suivant le code de la machine, (99,99%) le code ASCII.

Avantages :
- fichiers lisibles, par exemple avec un éditeur de texte

- fichiers portables (à peu près).

Inconvénients :
- travail de codage ralentissant l'écriture,
- encombrement plus important du fichier en moyenne.

Par exemple, soit la valeur entière 1024 sur 2 octets à écrire dans un fichier :

Fichier binaire :
Cette valeur en mémoire est codée en binaire :
$(00000100\ 00000000)_2$ soit
$(\quad 4 \qquad\qquad 0 \quad)_{10}$

Elle occupe 2 octets 4 et 0.

Fichier texte :

1024 est codée sur 4 char :
'1' '0' '2' '4' eux-mêmes codés par leur code ASCII :
49 48 50 52

L'ensemble occupe donc 4 octets.

Organisation d'un fichier

À côté des questions de codage, on parle d'organisation des informations. Les données d'un fichier étant généralement écrites ou lues par paquets, nommés enregistrements, on distingue les fichiers :
- à enregistrements fixes, les « paquets » font tous la même taille. Cela permet d'accéder facilement à n'importe quel paquets en sautant le nombre d'octets qu'il faut, un peut comme un tableau,
- à enregistrements variables, les « paquets » ne font pas tous la même taille. Dans ce cas seul un parcours séquentiel permet d'accéder à un enregistrement particulier.

Les avantages et inconvénients sont opposés : le premier sera plus « rapide » mais prends plus de place (pour bourrer les enregistrements plus petits), le second c'est le contraire.

Dans le cas où on accède à n'importe quel enregistrement d'un fichier, on parle d'**accès direct**, à l'opposé on parle d'**accès séquentiel**.

Gestion par le système, bufferisation

Le temps passé pour une opération sur un fichier ne dépend pas que de la taille de la donnée à accéder, il y a en quelque sorte une coût fixe et un coût variable. Les systèmes d'exploitation, pour limiter ces coûts accèdent généralement aux fichiers par l'intermédiaire de zone mémoires dédiées, appelés buffers ou caches, qu'ils remplissent de plusieurs octets d'un coup et exploitent avant de recommencer.

Remarque
C'est le système qui s'occupe des accès fichiers et qui décide quand les faire. C'est le cas des affichages à l'écran.

Quand vous avez un problème dans un programme, une technique d'analyse consiste à afficher des messages pour essayer de situer l'erreur. Or il est possible que le système décale l'affichage de vos messages et complique donc la recherche du problème.

Gestion en C++

Flux prédéfinies :

Il en existe plusieurs, les 3 principaux sont :

- cin
flux d'entrée associé au clavier par défaut. Cela est modifiable.
Ce flux est bufferisé, lorsque on tape sur le clavier, les données sont mises dans un tampon mémoire et fournies au programme lors des opérations de lecture futures.

- cout
flux de sortie associé par défaut à l'écran. Modifiable aussi.
Ce flux est bufferisé, lors d'une instruction d'affichage, les données à afficher sont mises dans un buffer et affichées à l'écran plus tard.

- cerr
flux de sortie associé par défaut à l'écran. Modifiable aussi.
Ce flux n'est pas bufferisé.

Utiliser cerr dans un contexte de recherche d'erreur plutôt que cout diminue le problème précédent sans l'éliminer toutefois.

Opérations de sortie :

Opérateur <<

sa syntaxe est :

```
flot_de_sortie << valeur
```

Son effet est d'envoyer la valeur vers le flot de sortie en la formatant de manière lisible.

Il est applicable à tous les types prédéfinis et surchargeables pour les classes qu'on définit.

Remarque, profitons-en pour rappeler la constante `endl` dont l'affichage provoque un renvoi au début de la ligne d'après.

Fonction membre write

Envoi au flot de sortie une suite de caractères de longueur donnée :

```
char T[100] = "Bonjour" ;
```

…

```
flot.write(T, 3) ;
```

envoie Bon vers flot

en particulier :

```
cout.write(T,3) ;
```

affiche Bon

Attention, il n'y a aucune vérification sur le nombre de caractères :

```
cout.write(T, 500) ;
```

produira certainement une erreur mémoire.

Il existe d'autres fonctions de sortie, mais ces 2 suffisent.

Opérations d’entrée :

Opérateur >>

sa syntaxe est :

flot_d_entree >> variable

Son effet est d’extraire une valeur du flot d’entrée et de la ranger dans la variable.

Une valeur est identifiée, séparée par des délimiteurs : espace, retour chariot, tabulation.

On parle d’espaces blancs !

Conséquence il n’est pas possible de saisir un espace blanc avec !

La chaîne :

Bonjour Léo

sera saisie en 2 fois !

Fonction membre get

Cette fonction extrait 1 caractère d'un flot d'entrée, elle est capable de lire les espaces blancs.

Exemple :

```
char C ;
cin.get(C) ;
```

Fonction membre getline

Elle a 3 arguments :
- 1 tableau de char S,
- 1 entier N,
- 1 caractère optionnel delim, valeur par défaut : retour à la ligne (‘\n’).

Elle extrait N - 1 caractères ou s’arrête si elle rencontre le caractère delim et elle ajoute le caractère ‘\0’ à la fin des caractères lus.

```
char T[100];

flot.getline(T, 80);
```

Attention, elle ne vérifie pas qu’il y a assez de place dans !

char T[100];

```
flot.getline(T, 200);          // Danger !
```

Fonction membre read

Symétrique de la fonction write.

Extrait du flot de d'entrée une suite de caractères de longueur donnée :

```
char T[100] ;
```

…

```
flot.read(T, 30) ;
```

Lit 30 caractères (octets)

Attention, il n'y a aucune vérification sur le nombre de caractères :

```
cout.read(T, 500) ;
```

produira certainement une erreur mémoire.

Fonction membre gcount

Renvoi le nombre de caractères effectivement lu par le dernier appel d'une fonction de lecture non formatée.

```
char T[100] ;

cin.getline(T, 80) ;

cout << "Nombre de caractères lus : " << cin.gcount() ;
```

```
...
bonjour (suivi de Entrée)
Nombre de caractères lus : 8
…
```

Le retour chariot est lu donc est compté !

Connexion d'un flot à un fichier

Si par défaut cin et cout (et cerr) sont connectés à leur flux respectifs, si on veut manipuler un fichier sur disque par exemple, il faut d'abord s'y connecter.

Et comme pour tout, il faut une variable pour le manipuler.

Le type d'une variable fichier se trouve dans la librairie fstream, qu'il faut donc inclure avant de commencer.

```
#include <fstream>
```

Déclaration d'une variable :

```
fstream F ;
```

Il existe en fait plusieurs type pour les fichiers permettant d'indiquer ce qu'on va faire avec :
- `ifstream` pour ne faire que des lectures,
- `ofstream` pour ne faire que des écritures,
- `fstream` pour faire les deux,

Branchement à un fichier, on parle **d'ouverture** d'un fichier, cela se fait avec la fonction membre open :

```
F.open (<Nom_du_fichier>, <mode>) ;
```

Le nom_du_fichier doit avoir du sens pour le système d'exploitation, par exemple :

`c:\utilisateurs\moi\mon-fichier.dat` pour windows

`.\data\mon-fichier.dat` pour windows

`/home/moi/mon-fichier.dat` pour linux

`./data/mon-fichier.dat` pour linux

…

Le mode indique au système d'exploitation ce que l'on va faire avec le fichier :

`ios ::in` : lecture du fichier

`ios ::out` : écriture dans le fichier

`ios ::app` : écriture à la fin du fichier

`ios ::binary` : le fichier est considéré comme binaire

Il est possible de combiner les modes avec | :

```
ios::in | ios::binary
```

Il est possible de combiner la déclaration et l'ouverture :

```
fstream F ;
F.open("/home/moi/mon-fichier.dat", ios::in) ;
```

est équivalent à :

```
fstream F("/home/moi/mon-fichier.dat", ios::in) ;
```

Fermeture d'un fichier

Lorsqu'on n'utilise plus un fichier et pour garantir que les modifications seront bien écrites sur le disque, il faut fermer un fichier :

```
F.close() ;
```

Test d'un fichier

Un fichier peut avoir des problèmes !

Il peut être absent, il peut être vide alors qu'on essaie de lire dedans, il peut être sur un support plein, etc.

Il faut donc tester l'état d'un fichier quand on s'en sert.

Test de la variable fichier :

```
fstream sortie("fichier.dat", ios::out) ;

il ( ! sortie )
    cout << "Problème à la création du fichier" << endl ;
```

si la valeur est != 0 (vraie) ça va,
si elle est = 0 (fausse) ça va pas.

Fonction de test :

Il existe aussi des fonctions qui donnent un (petite) idée de l'état du fichier :

eof() pour end of file :

s'active après une lecture qui n'a pas pu se faire parce que il n'y a plus rien à lire dans le fichier.

Exemple de lecture d'un fichier d'entiers jusqu'à la fin :

```
fstream F("mon-fichier.dat",ios::in) ;
int V, S = 0 ;

F >> V ;
while ( ! F.eof())
{
     S += V ;
     F >> V;
}
```

Autres fonctions de test :

Fonction good() : vraie si le fichier est en bon état !

Fonction fail() : vraie si problème en lecture, mais le problème est peut être réparable !

Fonction bad() : vraie si le fichier est planté !

Le test du flux est équivalent à la fonction good().

Accès direct dans un fichier

Un fichier est un paquets d'octets, pouvant être vide, mais quand il n'est pas vide, il est possible d'accéder directement à n'importe quel octet.

Pour accéder à un fichier tout se passe comme si il existait un pointeur qui pointe l'endroit (à l'octet près) où va se faire la prochaine opération d'écriture ou de lecture.

À l'ouverture, ce curseur est placé au début du fichier, et à chaque opération il avance du nombre d'octets concerné. si on lit un entier il avance de 2 ou 4 ou 8 octets en fonction de la taille d'un entier sur la machine, etc.

Il est possible de placer ce curseur où on veut. Et cela est possible indépendamment en lecture ou en écriture, en utilisant les 2 fonctions seekg et seekp :

`seekg(base, deplacement)` pour positionner le curseur de lecture (g pour get)
`seekp(base, deplacement)` pour positionner le curseur d'écriture (p pour put)

Ces 2 fonctions possèdent 2 arguments une base et un déplacement et le calcul est :

```
base + deplacement
```

Avec `base` qui peut être :
- `ios::beg` pour indiquer le début du fichier
- `ios::end` pour indiquer la fin du fichier
- `ios::cur` pour indiquer la position courante

Et `deplacement` qui est un entier.

Associées à ces fonctions on trouve les 2 fonctions qui permettent de savoir où se trouve les curseurs :

`tellg()` : donne la position du curseur de lecture
`tellp()` donne la position du curseur d'écriture

Utilisation classique :

```
fstream F("mon-fichier.dat",ios::in) ;

F.seekg(0, ios::end)

N = F.tellg();

N ?
```

Manipulateurs

Les manipulateurs permettent de modifier le formatage d'un fichier en sortie.

Par exemple vous connaissez la constante :

`endl`

qui renvoie à la ligne le prochain affichage.

Il en existe d'autres :

Dans `<iomanip>`

`setprecision(int N)` :
définit le nombre de chiffres après la virgule à afficher pour un flottant

```
double P = 3.14159;
cout << setprecision(2) << P;     // Affiche 3.14
```

setw(int N) :
définit la largeur du champ d'affichage

```
cout << 48 << endl << setw(8) << 35 << endl << "........" ;
```

Affiche :

```
48
      35
........
```

setfill(char c) :
définit le caractère de remplissage du champ d'affichage :

```
cout << setfill('+') << setw(8) << 35;
```

Affiche :

```
++++++35
```

Manipulateurs de format :

`dec`	écriture en décimal (défaut)
`hex`	écriture en hexadécimal
`oct`	écriture en octal
`left`	cadre à gauche dans le champ
`right`	cadre à droite
`boolalpha`	affiche les booléens true ou false
`noboolalpha`	affiche les booléens 1 ou 0

Vidange du buffer d'écriture :

```
flush
```

Les données contenues dans buffer sont écrites dans le fichier.

Utilisation :

```
flot << val << flush;
```

https://www.cplusplus.com/reference/iolibrary/