Langage C
Fichiers texte - version 0.01

1 Les fichiers

De point de vue du système UNIX, un fichier est une suite d'octets stockés sur un disque dur, mémoire SSD, clef USB ou tout autre support qui permet de préserver des données de façon permanente, sans aucune structure interne. L'ordre naturel d'accès aux données stockées dans le fichier est l'ordre séquentiel : on lit les données dans l'ordre, en commençant au début du fichier.
Nous allons distinguer deux types de fichiers : les fichiers textes, où chaque octet est un code de caractère [1], et les fichiers binaires où les données sont stockées sous forme brute.
Pour donner l'idée de la différence entre les deux types de fichiers supposons que nous voulions stocker un entier `int` - le résultat d'un calcul – dans un fichier. Pour stocker un `int` dans un fichier binaire, on copie simplement `sizeof( int )` octets de la mémoire vive (RAM) vers le fichier, sans aucune transformation.
Par contre, pour stocker le même entier dans un fichier texte, on le transforme en string et on copie ce string dans le fichier (mais sans le caractère nul à la fin).
Dans les deux cas, on écrit une suite d'octets dans un fichier, la différence est dans l'interprétation de ces octets.
Dans ce document, je commencerai à examiner les fichiers textes. Je supposerai que chaque octet d'un fichier texte est le code ASCII d'un caractère, *différent du caractère nul* (le caractère nul dans un fichier texte pose des problèmes à certaines opérations de lecture/écriture).
En général, le travail avec des fichiers passe par trois étapes :

1. l'ouverture de fichier avec `fopen`,
2. les opération de lecture et/ou écriture,
3. la fermeture avec `fclose`.

Toutes les opérations utilisées dans ce document demandent l'inclusion du fichier en-tête :

```
1 #include <stdio.h>
```

qui contient les prototypes de fonctions d'entrée/sortie.

2 Ouverture/fermeture de flot

```
1 FILE *fopen( const char *nom_fichier, const char *mode)
2 int   fclose( FILE *flot)
```

1. Pour nous, ce sera toujours le code ASCII. Mais en C il existe aussi un format "wide characters" `wchar_t`.

`fopen` ouvre un fichier `nom_fichier` et retourne un flot (un stream, un point d'accès au fichier) de type *FILE ** – ou `NULL` en cas d'échec de cette ouverture.
Le paramètre `mode` sert à spécifier le mode d'ouverture.
Le tableau ci-dessous présente les 6 modes d'ouverture existant en C :

	`"r"`	`"w"`	`"a"`	`"r+"`	`"w+"`	`"a+"`
le fichier doit exister	oui			oui		
le contenu du fichier est écrasé à l'ouverture		oui			oui	
lecture possible	oui			oui	oui	oui
écriture à la position courante		oui		oui	oui	
écriture à la fin du flot (mode append)			oui			oui

Un « oui » dans une colonne du tableau indique une condition d'ouverture, une propriété du flot ou l'état du flot.
Notez que, pour tous les modes sauf `"r"`, il y a un « oui » dans une de deux dernières lignes : cela signifie que pour tous ces modes le fichier est ouvert en écriture. Par contre, l'écriture ne se fait pas toujours à la même position : pour les modes avec `a` (append) la position courante du fichier [2] se déplace à la fin de fichier avant chaque opération d'écriture.

Exemples.

```
1 FILE *flot=fopen("toto.txt", "r+");
```

Les « oui » à l'intersection de la colonne `"r+"` et de la ligne `"le fichier doit exister?"` indique que l'opération réussit uniquement si le fichier `"toto.txt"` existe. Si le fichier n'existe pas, l'ouverture échoue et `fopen` retourne `NULL`. Si l'ouverture réussit, nous pouvons effectuer aussi bien des opérations de lecture que d'écriture. De plus l'écriture dans le fichier s'effectue à partir de la position courante.

```
1 FILE *flot=fopen("toto.txt", "w+");
```

Le fichier `toto.txt` sera ouvert en lecture et en l'écriture. A l'intersection de la ligne `"le contenu du fichier écrasé à l'ouverture"` et de la colonne `"w+"` on trouve « oui » qui signifie que si le fichier existe à l'ouverture le contenu sera effacé, c'est-à-dire la taille de fichier devient 0.
Il n'y a pas « oui » à l'intersection de `"w+"` et de la ligne `"le fichier doit exister ?"` donc cette opération réussit même si le fichier `toto.txt`. `fopen` créera un nouveau fichier initialement vide.

Avertissement. Le dernier paramètre de `fopen` est un string donc il ne faut jamais écrire `fopen("toto.txt", 'r')` mais toujours `fopen("toto.txt", "r")`.

L'opération `fclose` ferme le flot.

2.1 Flots `stdin`, `stdout`, `stderr`

Trois flots sont ouverts dès le début de l'exécution du programme, sans qu'il soit nécessaire de les ouvrir avec `fopen` :

- `stdin` – flot d'entrée standard, permettant la lecture depuis le terminal,
- `stdout` – flot de sortie standard, écriture sur le terminal,

2. La notion de la position courante sera expliquée en détail ci-dessous.

– `stderr` – flot de sortie d'erreur standard, écriture sur le terminal.

Il est possible de rediriger les trois flots vers des fichiers au moment du lancement du programme depuis le terminal :

```
./myprogram  < infile > outfile 2> errfile
```

redirige `stdin` vers le fichier `infile`, `stdout` vers le fichier `outfile` et `errfile` vers le fichier `errfile`. Donc, par exemple, la lecture sur le flot `stdin` va s'effectuer depuis le fichier `infile`. Bien évidemment, il est possible de rediriger seulement un ou deux des flots au lieu des trois.

Il faut souligner que les trois flots sont des objets de type `FILE *` et peuvent être utiliser dans (presque) toutes les fonctions qui prennent `FILE *` comme paramètre.

Examinons le fragment de code :

```c
/*   prog.c   */
int main(int argc, char **argv){
  FILE *out;
  if( argc == 2 )
     out = fopen( argv[1], "w" );
  else
     out = stdout;
  /* faire toutes les opérations de sortie sur le flot out */
```

Si on lance ce programme par une commande de la forme `./prog file`, toutes les écritures sur le flot `out` seront envoyées vers le fichier `file`. Par contre, si on lance le même programme sans paramètres – `./prog` – les données écrites sur le flot `out` seront envoyées vers le terminal.

3 Lecture et écriture caractère par caractère

```c
int fgetc(File *flot)
int getc(File *flot)
int getchar(void)
int putc(int c, FILE *flot);
int fputc(int c, FILE *flot)
int putchar(int c)
```

`fgetc` est une fonction, tandis que `getc` est une macro-fonction [3]. `getchar()` est équivalent à `getc(stdin)`.

En cas de succès, les fonctions `fgetc`, `getc`, `getchar` retournent le caractère lu (sous forme d'un `int` et non pas un `char`). Arrivé à la fin du fichier ou en cas d'erreur, elles retournent la constante prédéfinie `EOF`.

Il est important d'affecter la valeur retournée par ces fonctions dans une variable `int` et non pas dans une variable `char`. En effet, la constante `EOF` est un `int` et le code

```c
//  incorrect, le résultat de fgetc() doit aller dans un int
char c = fgetc( flot );
if( c == EOF ){
```

3. La différence entre les fonctions et les macro-fonctions ne nous concerne pas pour l'instant.

```
// gérer fin de fichier ou erreur

}
```

risque de ne pas détecter la fin de fichier.
Le code correct :

```
FILE *f = fopen( fichier, "r");
int i, count = 0;
while( ( i = fgetc( f ) ) != EOF ){
  count ++;
  char c = i;
  // traiter le caractère c
}
```

Rappelons qu'on peut fermer l'entrée standard attachée au terminal avec `Ctrl-D`. Cela a le même effet que l'atteinte de la fin d'un fichier normal, c'est-à-dire une fois `stdin` fermé par `Ctrl-D` `getchar()`, `fgetc(stdin)` et `getc(stdin)` retournent `EOF`.

4 Lecture/écriture par ligne

```
char *fgets( char *buf, int i, FILE *flot)
```

La fonction `fgets` lit les caractères du `flot` jusqu'à ce que :
- le caractère de nouvelle ligne `'\n'` soit lu, ou,
- `i - 1` caractères soient lus, ou,
- la fin du flot soit atteinte.

La lecture s'arrête dès que l'une de ces trois condition est satisfaite. Les caractères lus sont placés à l'adresse `buf`, et le caractère nul est ajouté à la fin. `fgets` retourne l'adresse `buf` si au moins un caractère a été lu. Si aucun caractère n'a été lu (fin du flot) ou en cas d'erreur, `fgets` retourne `NULL`.

```
int fputs (const char *s, FILE *flot)
int puts(const char *s)
```

`fputs` écrit dans le `flot` tous les caractères de string `s`, sauf le caractère nul qui termine `s`.

`puts` écrit dans `stdout` tous les caractères de string `s`, sauf le caractère nul, suivis du caractère de nouvelle ligne `'\n'`.

Notez que `puts("toto")` **n'est pas équivalent** à `fputs("toto", stdout)` puisque `puts` écrira dans `stdout` les caractères `toto\n` tandis que `fputs` écrira seulement `toto`.

Exemple. Une tâche typique dans un programme interactif consiste à poser une question sur le terminal et attendre une réponse de l'utilisateur.
Par exemple on affiche

`"[y]es or [n]on : "`

sur le terminal et le programme attend que l'utilisateur entre soit `y` soit `n`. L'idée, naïve et incorrecte, est d'utiliser la boucle suivante où en attend que l'utilisateur répond soit `y`, soit `n`, soit ferme `stdin` avec `Ctrl-D` :

```c
int i  ;
char *s = "[y]es or [n]on : ";

do{
  fputs( s , stdout);
  i = getchar();
  if( i == 'n' || i == 'y' || i == EOF )
    break;
}
while( 1 );

switch( i ){
  case EOF:
    puts("\nEOF\n");
    break;
  case 'y' :
    puts("\nyes\n");
    break;
  case 'n':
    puts("\nno\n");
    break;
  default :
    break;
}
```

On lance le programme mais au lieu de répondre `y` ou `n` on répond `v` :

```
$ ./a.out
[y]es or [n]on : v
[y]es or [n]on : [y]es or [n]on :
```

On constate qu'après une mauvaise réponse le texte `"[y]es or [n]on :"` s'affiche deux fois. Avec `gdb`[4] on découvre que `getchar` lit deux caractères : d'abord `v` et ensuite `'\n'`. Ce dernier caractère est ajouté dans `stdin` parce que après `v` nous avons tapé ENTER, ce qui envoie `'\n'` vers `stdin`.
Le code correct doit prendre en compte toutes les erreurs possibles de l'utilisateur (intentionnelles ou non) comme par exemple :

- l'utilisateur fait ENTER tout de suite, sans avoir entré aucun autre caractère,
- il entre une suite de plusiers caractères avant de faire ENTER. On supposera que le dernier caractère dans la suite est la réponse, donc `abcy ENTER` correspond à `y ENTER`.

Le code suivant est plus robuste que le précédent :

```c
int i, j  ;
char *mes = "[y]es or [n]on : ";
do{
  fputs( mes, stdout);
  //recupérer le premier char
```

4. j'espère que tout le monde utilise `gdb` pour deboguer son programme

```
  i = getchar();
  if( i == EOF ) //Ctrl-D
    break;
  if( i == '\n' ) //ENTER au début de la ligne
    continue;     //revenir à fputs

    //faire defiler les caracteres jusqu'à la nouvelle ligne
    //mémoriser le caractère précédent dans i
  j = i;
  do{
    i  = j;
    j = getchar();
  }while( j != '\n'  && j != EOF );

  if( j == EOF ){
    i = EOF;
    break;
  }

}while( i != 'n' && i != 'y'  );

/* switch comme dans le précédent */
```

5 Position courante

Chaque flot possède une position courante. Il est commode de voir cette position courante comme une position entre deux octets : position 0 avant le premier octet du flot, position 1 entre le premier et le deuxième octet etc.
Toute lecture se fait à partir de la position courante, et la lecture avance cette position immédiatement après le dernier octet lu.
Pour les écritures, il faut distinguer deux situations.
Dans le mode *append* (mode `"a"` ou `"a+"` comme deuxième argument de `fopen`), chaque opération d'écriture commence par déplacer la position courante à la fin du flot. Donc tous les octets écrits sont automatiquement ajoutés à la fin du fichier. Après l'écriture la position courante est immédatement après le dernier caractère écrit, donc encore à la fin de flot.
Dans tous les autres modes qui permettent écriture, l'écriture s'effectue à partir de la position courante. Après l'écriture, la position courante est immédiatement après le dernier caractère écrit.
Exemple. On suppose que le fichier `toto.txt` contient les caractères suivants : `abcdefghij`.

```
FILE *f = fopen( "toto.txt", "r+" );
char *c;
int i;

i = fgetc(f); //lit 'a'
i = fgetc(f); //lit 'b'
fputc( 'x', f); //remplace 'c' par 'x'
```

```
fputc( 'y', f); // remplace 'd' par 'y'
i = fgetc( f ); //lit 'e'
```

Pour comparer, regardons ce qui se produit pour le même fichier en mode `"a+"` :

```
FILE *f = fopen( "toto.txt", "a+" );
char *c;
int i;

//même en mode append, initialement la position courante est  0
i = fgetc(f); //lit 'a'
i = fgetc(f); //lit 'b'
// fputc commence par déplacer la position courante à la fin de flot
fputc( 'x', f); // écrit  'x'  après 'j'
fputc( 'y', f); // écrit  'y'  après 'x'
i = fgetc( f ); // retourne EOF
```

Il est possible de changer la position courante sans effectuer de lecture ou d'écriture, en utilisant :

```
int fseek(FILE *flot, long offset, int whence)
```

Le paramètre `whence` peut prendre une de trois valeurs :

- `SEEK_SET` – position au début du flot,
- `SEEK_CUR` – position courante,
- `SEEK_END` – position à la fin du flot.

Le paramètre `offset` spécifie le décalage par rapport à la position donnée par `whence`.

Exemple.

- `fseek( flot, 20L, SEEK_SET);` place la position courante 20 octets après le début du flot,
- `fseek( flot, -20L, SEEK_END);` place la position courante 20 octest avant la fin du flot,
- `fseek( flot, -10L, SEEK_CUR);` fait reculer la position courante de 10 octets.

Il est impossible d'ajouter des caractères avant le début du fichier. Par exemple, il est **impossible** de se positionner avant le début du fichier :

```
if( fseek( flot, -1L, SEEK_SET) < 0 )
   perror("fseek");
```

affiche l'erreur `invalid argument`.

Il est tout-à-fait possible d'ajouter des caractères après la fin d'un fichier.

```
fseek( flot, 10L, SEEK_END);
fputc( 'x', flot) ;
```

Le caractère `'x'` est ajouté 10 octets après la fin de fichier. Le « trou » de 10 octets entre l'ancien fin de fichier et le nouveau caractère sera rempli par le caractère nul (ce qui est nuisible dans un fichier texte).

```
long ftell( FILE *flot)
```

La fonction `ftell` retourne la position courante.
Donc, pour trouver la longueur du fichier, il suffit de se déplacer à la fin avec `fseek` puis d'appeler `ftell`. (Ce n'est pas la meilleur méthode pour trouver la longueur de fichier, en cours de Programmation systèmes vous apprendrez comment le faire de manière plus propre dans les systèmes UNIX/Linux.)

Pour les fichiers très très longs (plusieurs Giga), si longs que l'entier de type `long int` n'a plus la capacité de stocker la longueur de fichier, les fonctions `fseek` et `ftell` sont inutiles. Les fonctions `fgetpos` et `fsetpos` permettent de gérer la position courante pour de très longs fichiers. Pour les détails voir les pages man de ces fonctions.

Le programme peut ouvrir un même fichier plusieurs fois en créant plusieurs flots. Chaque flot gère sa position courante de façon indépendante.
Par exemple, pour changer dans un fichier texte toutes les lettres minuscules en majuscule on peut procéder de façon suivante :

- on ouvre le fichier deux fois, la première fois en mode `"r"` ce qui donne un flot `in`, et la deuxième fois en mode `"r+"` ce qui donne le deuxième flot `out`,
- dans une boucle
  1. on lit le flot `in` en mettant les caractères lus dans un tampon `buf`,
  2. on applique `toupper` à tous les caractères de `buf`,
  3. on écrit le contenu de `buf` dans le flot `out`.

6 Tampon de flot

Les opérations de lecture/écriture de fichiers sont très coûteuses en temps. Cela est dû à deux raisons :

1. Le programme n'accède pas directement aux fichiers. Quand le programme fait une lecture/écriture de fichier, il demande en fait au noyau de système d'exploitation de faire cette opération pour lui. L'appel au service du noyau est bien plus compliqué que l'appel d'une fonction, et demande plus de temps.

   Pendant que le noyau lit le fichier, l'exécution du programme est suspendue : c'est le code qui réside dans le noyau du système qui est exécuté. Une fois l'opération terminée, le noyau rend la main au programme. Cela aussi prend plus de temps qu'un retour de fonction.
2. L'accès à la mémoire sur un disque ou une carte SSD prend beaucoup de temps, bien plus de temps que l'accès à la mémoire vive (RAM).

Pour diminuer le nombre d'accès au fichier, le flot maintient un tampon (buffer).
Par défaut, juste après l'ouverture de flot avec `fopen` le tampon de flot est géré en mode *fully buffered*. Cela signifie que les écritures dans le flot sont en fait effectuées dans le tampon, et c'est seulement quand le tampon est plein que son contenu est entièrement et d'un seul coup écrit dans le fichier.
Pour la lecture, la première lecture, au lieu de lire par exemple un seul caractère, lit une zone de fichier de taille égale à la taille de tampon. Les lectures suivantes fournissent au programme les caractères depuis le tampon, sans solliciter le noyau. C'est seulement quand le programme demande des caractères qui ne sont pas dans le tampon de lecture qu'il y a un nouvel accès au ficher, qui remplit à nouveau le tampon.

Il existe aussi la possibilité d'associer au flot un tampon *line buffered*. Dans ce cas, l'écriture dans le fichier se fait quand le tampon reçoit le caractère de nouvelle ligne.
Finalement, il est possible d'avoir des flots *unbuffered*, c'est-à-dire sans tampon.
La gestion de tampon s'effectue à l'aide de la fonction `setvbuf`.
L'appel à la fonction

```
1 int fflush( FILE *flot )
```

force l'écriture de tampon associé au `flot` dans le fichier.
`fflush(NULL)` force l'écriture de tampons de tous les flots ouverts.
L'appel à `exit` et `return` de `main` provoquent automatiquement l'écriture de tous les tampons de flots dans les fichiers.
Le flot `stderr` est *unbuffered* `stdin`, `stdout` sont buffered.