CHAPITRE

7

Entrées et sorties en C

7.1 Lire et écrire des données dans des fichiers personnels

Jusqu'ici les programmes dont nous avons discutés ne concernent que des e/s pré-définies, standard. Nous voulons lire et écrire dans les fichiers usuels de l'utlisateur, non pas ceux du système.
Pour accéder à un fichier privé, le C définie une structure de données appelée FILE pour interfacer logiciel et espace disque.

> `struct FILE` : ses champs contiennent les informations pertinentes sur le fichier visé.

Le nombre et le nom des champs de FILE peuvent varier d'un SE à un autre. Toutefois nous reconnaitrons les champs suivant

```
struct FILE {
char *buffer;  → pointe vers l'adresse du tampon
char *ptr;  → début du tampon
int cnt;  → nombre de caractères du tampon (dimension)
int fd;  → identifiant
etc ...
```

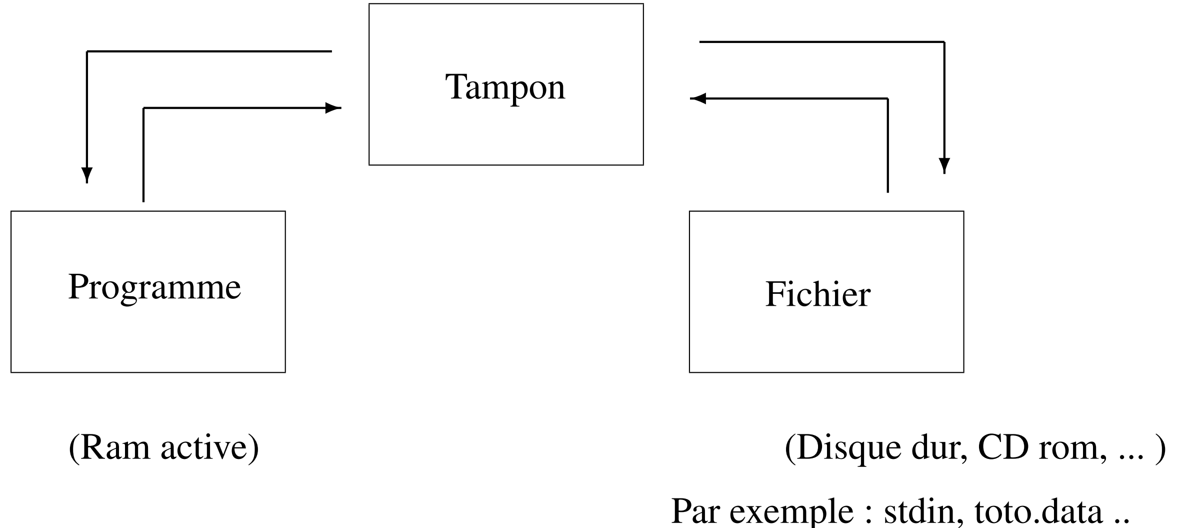
Quelques remarques :

- `struct FILE` : déclaré dans `stdio.h`
- les fonctions accédant aux fichiers : `type FILE` (sauf avis contraire)
- ces fonctions sont toutes standards

7.2 Gestion des entrées/sorties : notion de tampon

Un programme C n'accède pas directement aux données emmagasinées sur le disque dur, pour des raisons d'optimisation : taux de transfert (net) faible, perte d'efficacité en travail partagé (plusieurs utilisateurs). Le **tampon** est un espace mémoire intermédiaire (un *swap*) où sont empilées les données du programme avant d'être transférées au disque lors de l'écriture ; la même situation se présente lors de la lecture d'un fichier sur le disque (voir Figure 1).

Figure 1 - Diagramme du flot de données entre logiciel et disque. La zone tampon est temporaire mais essentielle.

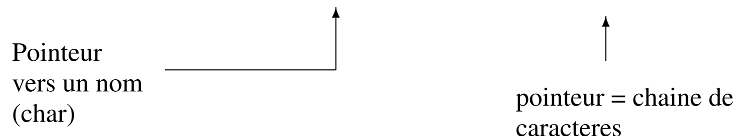


Il faut donc distinguer entre le temps où les données sont transférées par le logiciel C (c'est-à-dire le point d'exécution dans le programme) et le temps réel où les données apparaissent sur le disque. Cet intervalle de temps peut mener à des pertes de données si, par exemple, le SE devait être interrompu au moment de l'exercice d'E/S. On pourra palier à ces problèmes grâce à des fonctions spécifiques.

7.3 Ouverture et fermeture de fichiers

L'accès aux fichiers est formalisé par les fonctions *fopen* et *fclose*. L'ouverture se fait par un appel à *fopen* selon la syntaxe

Ouvrir un fichier : FILE *fopen (identifiant FILE, char *mode)



Par exemple, les déclarations

```
char *nom = ``toto.data'' ;
char *mode = ``r'' ;
FILE *fp;
```

permettraient au programme d'accéder *en mode* lecture le fichier toto.data comme ceci :

```
fp = fopen( nom, mode ) ;
```

Le résultat de *fopen* est donc affecté au pointeur *fp*. Comme sous UNIX/LINUX, les fichiers sont lus suivant un mode d'accès : ce mode peut être un des suivants :

Mode	signification
r	lecture, si fichier existe (r = read)
r+	lecture et écriture
w	écriture (w = write)
w+	écriture et lecture, fichier créé si n'existe pas
a	"append" : écrire à la fin
a+	comme 'a', mais crée fichier s'il n'existe pas

TAB. 7.1 – Modes d'accès des fichiers en C

La fermeture d'un fichier : permet d'éviter des erreurs d'entrée/sortie, et notamment d'éviter les pertes de données. Un fichier peut être fermé aussi simplement qu'il est ouvert par appel à la fonction *fclose* dont la syntaxe est

```
int fclose( FILE *pointeur ) ;
```

Remarquez que *fclose* retourne une valeur entière. On peut faire l'affectation suivante :

```
int n ;
n = fclose( fp ) ;
if( n == 0 ) printf( `` Fermeture r\'eussie '' ) ;
```

car $fclose = 0$ si l'opération a réussi. Cette opération de fermeture assure que le tampon est **vidé** de toute donnée résiduelle non-encore transférée au fichier (par exemple, en mode W [écriture]).

7.4 Quelques opérations de lecture/écriture

Les fonctions

`fputc, fputs; fgetc, fgets`

sont les équivalents non-formattés des fonctions d'E/S sur fichiers standards. Ainsi toujours avec *fp* le pointeur vers la structure FILE de tout à l'heure :

1. En mode caractère : fgetc admet la syntaxe

   ```
   int fgetc( fp );
   ```

   → lit dans fichier pointé par fp un et un seul caractère quel qu'il soit

   → si `fgetc()` retourne comme valeur EOF (constante symbolique 'end-of-file') : fin de fichier détectée.

2. Toujours avec les caractères : `fputc` fait l'opération inverse. Sa syntaxe

   ```
   int fputc( int, fp );
   ```

   où un entier (int) serait converti en caractère puis écrit dans le fichier vers lequel pointe `fp`.

3. En mode chaîne (string) : permet de lire toute une chaîne, tous ses caractères, sans interruption. Syntaxe :

   ```
   char *fgets( char *pt, int, fp );
   ```

   Ici le premier argument est un pointeur vers un espace mémoire (pour le stokage des données lues) ; l'argument suivant 'int' est un entier indiquant la longueur maximale de la chaîne permise ; et *fp* pointe vers un fichier particulier.

   Quelques remarques sont de mise :

   – en lecture fgets ajoute le symbole caractère null '\0' automatiquement.
   – lecture s'arrête soit a) quand le nombre max de caractères est atteint b) un caractère de changement de ligne est rencontré ('\n') c) fin de fichier détectée

   La fonction fgets() retourne une adresse mémoire, soit celle de la chaîne qui contient les données lu. Si une erreur de lecture se présente alors fgets() = NULL, ce qui nous permettra de détecter des erreurs.

4. Écriture de chaînes : l'opération inverse à fgets() se fait par appel à la fonction `fputs()` dont la syntaxe est

   ```
   char *fputs( char *pt, fp );
   ```

c'est-à-dire que le pointeur 'pt' est initialisé pour identifier la chaîne à sauvergarder.

7.5 Lecture/écriture formattées

Les fonctions `printf` et `scanf` que nous connaissons bien et qui portent à l'utilisation de fichiers standards (*stdout* et *stdin* respectivement) sont une application spéciale des fonctions

`fprintf, fscanf .`

Toujours avec *fp*, le pointeur de type structure FILE, nous aurons la syntaxe

int fprintf(fp, déclaration de formats, arguments)

pour `fprintf()` : si la valeur (entière) retournée = EOF ou <0, une erreur se présente. Les formats et arguments sont identiques à l'appel de `printf()`. En fait

fprintf(stdout, formats, arguments)

et

printf(formats, arguments)

sont la seule et même opération. En lecture nous aurons

fscanf(stdin, formats, arguments)

et

scanf(formats, arguments)

comme opérations identiques.

7.5.1 Se positionner dans un fichier

Essentiellement, nous voulons pouvoir reprendre la lecture d'un fichier sans avoir à effectuer l'opération de fermeture et de ré-ouverture de ce fichier. La fonction `rewind` permet cela. Sa syntaxe

```
void rewind( fp );
```

permet de se placer au début du fichier ouvert et identifié par *fp*. Note : cela suppose que l'opération d'ouverture a été possible.

On peut se déplacer relativement à un point de référence ou son point actuel dans le fichier courant. La fonction *fseek()* permet de contrôler ses déplacements en octets. Sa syntaxe

int fseek(fp, long int, int) ;

permet de donner un déplacement positif ou négatif en argument (long int). La position de référence se distingue ainsi

0 : début du fichier. Constante symbolique `SEEK_SET` est identique

1 : Position courante. Constante symbolique `SEEK_CUR` est identique.

2 : fin du fichier ou encore `SEEK_END`.

Pour obtenir sa position actuelle dans un fichier il suffit d'évoquer la fonction `ftell()` qui retourne la valeur (en octets) de son déplacement relativement au début du fichier. Sa syntaxe est simplement

```
int ftell( fp );
```

Comme avant, le nom du pointeur prendra une valeur permise (fichier ouvert avec succès). Les erreurs d'opération seront reconnues par les valeurs retournées par `fseek()` comme indiqué ici.

7.5.2 Saisir des paramètres à la ligne de commandes

Jusqu'ici nous avons toujours fait la saisie de données interactivement sur l'entrée standard au moyen de scanf. Nous pouvons par définition de la fonction `main()` d'un programme lui donner des arguments, comme à toute autre fonction.
Contrairement aux autres fonctions, les arguments de `main` seront lus sur la ligne de commande et non pas d'un point du programme à un autre. Par exemple, nous avions toujours initialisé une variable à un point du programme avant de la passer en argument dans l'appel d'une fonction. Pour les arguments de main, l'initialisation doit se faire à partir de ce qui est entré au clavier : il faut donc faire un balayage du fichier d'entrées standard, *stdin*. La syntaxe du C sera ainsi :

```
void main( int argc, char *argv[] ) { ... }
```

On peut imaginer l'argument *argv* comme une chaîne de caractères qui contiendra les paramètres passés sur la ligne de commande. Le premier argument, l'entier *argc*, sert à compter les différents paramètres passés à *main*. Ils seront comptés selon qu'un espace vide apparait dans *argv* ou non.
Lorsque l'on désire faire l'affectation des entrées contenues dans *argv*, on peut leur faire référence comme dans un tableau, et en tenant compte du type (caractère) de argv.

L'appel d'un progamme prenant deux arguments, de type caractère et entier, se fera comme pour une commande Unix. Ainsi

```
a.out mon_nom 22
```

pour saisir le nom et l'âge du professeur donnerait en tout **trois** arguments sur la ligne de commande (argc = 3), car le nom du programme figure à la liste des arguments saisis. Il ne reste plus, dans le programme, qu'à découper *argv* pour faire l'initialisation des variables pertinentes.
Par exemple, on pourrait faire l'initialisation d'une chaîne de caractères appelées toto par

```
toto = argv[1];
```

si le premier argument du tableau argv lui correspond. Si le deuxième argument est un nombre entier ? Il faut alors convertir le caractère saisi par appel d'une fonction spéciale, `atof`.
Ainsi avec la déclaration `int i;` nous pourrions faire l'affectation suivante

```
i = atof( argv[2] );
```

si le deuxième argument du programme doit bien correspondre à un nombre entier. Remarquez que *atof()* convertie d'abord en flottant le ou les caractères ASCII saisi représentant le nombre, et c'est l'affectation (=) à la variable qui opère un cast implicite et convertie le flottant en entier.