Programmation impérative

Cours 4 : Manipulation des fichiers en C

Catalin Dima

Organisation des fichiers

- Qqs caractéristiques des fichiers :
  - Nom (+ extension).
  - Chemin d'accès absolu = suite des noms des répertoires qu'il faut traverser, en partant du répertoire racine, pour trouver le fichier.
  - Chemin d'accès relatif = tout programme est associé à un répertoire de travail (hérité lors du lancement en exécution), alors on peut retrouver les fichiers en partant de ce répertoire.
  - Taille.
  - Propriétaire du fichier.
  - Droits et modes d'utilisation : accessible en lecture/écriture, exécutable, archivé.
  - Presque toujours, les droits et modes d'utilisation sont associés à des (groupes d') utilisateurs.
- Dans un fichier, les données sont stockées de manière séquentielle sur les espaces de stockage.
  - Différents modes d'organisation interne existent, suivant le système de fichiers utilisé (NTFS, FAT, ext3, etc.).

Utilisation programmatique des fichiers

- Manipulation des grandes structures de données (listes très grandes) qui ne peuvent pas être stockées dans la mémoire du programme.
- Sauvegarde des calculs à réutiliser dans des exécutions ultérieures.
- Enchaînement des entrées-sorties entre plusieurs programmes : les résultats d'un programme sont redirigées en entrée d'un autre programme.
- Par extension, toute entrée et toute sortie d'un programme est considérée comme étant faite dans un fichier :
  - stdin/stdout pour les entrées au clavier et sorties à l'écran.
  - Fichiers spéciaux (tubes, files d'attente) pour les entrées-sorties entre les processus sur un système d'exploitation, ou pour les entrées-sorties à travers les interfaces réseau (sockets).

Ouverture d'un fichier en C

- Structure `FILE` permettant la manipulation du fichier.
  - Déclarée dans `stdio.h`.
  - Les détails de cette structure sont dépendants de la plate-forme d'exécution (système d'exploitation, compilateur).
  - Les fichiers ne sont jamais manipulés en accessant les champs de cette structure, mais à travers des fonctions spécifiques qui utilisent des pointeurs de `FILE` comme argument !
- Pour manipuler des fichiers il faut toujours déclarer des variables `FILE *`.
- Ouverture d'un fichier : fonction `FILE* fopen(char chemindacces[], char mode[])`.
- Premier exemple :

```
FILE* monfis;   // variable qui servira à manipuler le fichier
monfis = fopen("fichier.txt","r");   // ouvert en lecture seulement !
```

- Lors d'un appel comme `fopen("fichier.txt","r")`, le fichier est cherché dans le répertoire de travail ! (chemin d'accès relatif)

Types d'ouverture de fichiers

- Premier argument de `fopen` : chemin d'accès relatif ou absolu.
  - Sous linux, tout argument qui commence par `/` est un chemin d'accès absolu.
  - Sous windows, les chemins absolus commencent par une lettre de "volume", les séparateurs des noms de répertoires étant écrits en double (car l'antislash a une interprétation particulière dans une chaîne de caractères !).
- Deuxième argument : un nombre restreint de chaînes de caractères :
  - `fopen("fichier.txt", "r")` : fichier ouvert en lecture seule, le pointeur de position courante est placé en début de fichier. Erreur si fichier inexistant.
  - `fopen("fichier.txt", "r+")` : fichier ouvert en lecture/écriture, le pointeur de position courante est placé en début de fichier. Fichier créé si inexistant.
  - `fopen("fichier.txt", "w")` : fichier ouvert en écriture par écrasement, le pointeur de position courante est placé en début de fichier. Le fichier est créé si inexistant, tronqué à 0 si existant.
  - `fopen("fichier.txt", "a")` : fichier ouvert en lecture/écriture, le pointeur de position courante est placé en fin de fichier.
  - Deux autres possibilités : `"w+"`,`"a+"`.
- `fopen` renvoie `NULL` si le fichier n'a pas pu être ouvert, par exemple :
  - Erreur car fichier inexistant qu'on veut ouvrir en `"r"`.
  - Fichier inaccessible en lecture ou en écriture (alors voir l'affichage de `ls -l` !)
  - Fichier bloqué en lecture ou écriture car utilisé par un autre programme.

Manipulation des fichiers

- `fscanf(FILE*,...)` : lecture des données.
- `fprintf(FILE*,...)` : écriture des données.
  - Premier argument : le pointeur `FILE*` identifiant le fichier dans lequel l'opération sera exécutée.
  - Le reste des arguments sont similaires aux `scanf/printf` : une chaîne de caractères formattant l'opération, puis une liste d'adresses de variables (pour `scanf`) ou d'expressions (pour `printf`).
- Exemple :

```
FILE* monfis;
int a,b;
monfis = fopen("/home/dima/fichier.txt","r");
fscanf(monfis,"%d%d",&a,&b);
     // fichier.txt doit avoir la structure suivante, sinon erreur :
 // deux entiers séparés par plusieurs espaces (caractères)
```

- Autres fonctions : `int fgetc(FILE* fichier)`, `char *fgets(char *s, int size, FILE * fichier)`.

Pointeur de position de fichier

- Toute lecture ou écriture d'une donnée dans le fichier se fait à l'endroit pointé par le *pointeur de position de fichier*.
  - Il est caché dans la structure `FILE`, mais pas besoin de le manipuler directement.
- Toute lecture/écriture d'une donnée avance ce pointeur du nombre d'octets lus/écrits.
  - Cas le plus simple : on lit des caractères avec `fgetc(monfis)` ou `fscanf(monfis,"%c",car)` ;
  - Avec chaque appel à `fgetc`, le pointeur de fichier avance d'une position vers la fin du fichier.
  - Pour une lecture d'entier avec `fscanf(monfis,"%d",entier)`, le pointeur de fichier avance du nombre d'octets utilisés pour représenter l'entier en décimal, plus un séparateur.
  - Mêmes règles pour l'écriture avec `fprintf`, en fonction du format choisi d'écriture.

Pointeur de position de fichier (2)

- Exemple :

```
FILE* fich;
char c[20];
int a;
fich = fopen("donnees.txt","r+");
c[0]=fgetc(fich);
fscanf(fich,"%d",&a);
fprintf(fich,"%d",a-100);
   // si dans le fichier on avait c120 234
 // après ces 2 opérations on aura c120 204
```

Opérations sur le pointeur de position de fichier

- `int fseek(FILE *fich, long decalage, int parrapport)` : déplacer le pointeur de fichier à la position `offset` par rapport à la place indiquée par `parrapport`.
- Trois valeurs admises pour `parrapport` :
  - `parrapport = SEEK_SET` ou 0 : par rapport au début du fichier.
  - `parrapport = SEEK_CUR` ou 1 : par rapport à la position courante dans le fichier.
  - `parrapport = SEEK_END` ou 2 : par rapport à la fin du fichier.
- Exemples :
  - `fseek(fich,0L,SEEK_SET)` : se mettre au début du fichier.
  - Macro de raccourci avec le même effet : `rewind(FILE*)`.
  - `fseek(fich,0L,SEEK_END)` : se mettre à la fin du fichier.
  - `fseek(fich,-1L,SEEK_CUR)` : faire reculer d'une position le pointeur de position.
- `long ftell(FILE *fich)` : retrouver la position du pointeur dans le fichier par rapport au début.

Autres fonctions

- `int fclose(FILE* fich)` : fermer un fichier.
  - Pourquoi fermer ? parce que les opérations peuvent s'exécuter avec un décalage.
  - Et il se peut que, lorsqu'on termine le programme, certaines opérations ne soient pas terminées, terminées, donc risque de perte des données !
  - `fclose` "tire la chasse d'eau", toute opération dans le fichier est exécutée, puis le fichier détaché du programme.
- `short feof(FILE* fich)` : renvoie non-zéro si la fin du fichier est franchie.
  - Très utile si on veut avoir une condition d'arrêt par exemple lorsqu'on affiche toutes les données du fichier !
- `short fputc(char c, FILE* fich)` : écrire un caractère dans `fich`.
- `short fputs(char c[], FILE* fich)` : écrire une chaîne de caractères dans `fich`.

Autres fonctions (2)

- `unsigned short fread(void *ptr, unsigned short nbre, FILE* fich)` : lire une suite de `nbre` octets à partir de la position courante dans `fich` et les mettre dans le pointeur `ptr`.
  - Attention ! Le pointeur `ptr` doit pointer soit un tableau correctement déclaré, soit avoir été alloué précédemment !
  - `fread` renvoie le nombre d'octets effectivement lus (car il se peut qu'on franchisse la fin de fichier avant de lire `nbre` octets !).
- `unsigned short fwrite (const void *ptr, unsigned short size, unsigned short nbre, FILE *fich)` : le dual de `fread`, écrit dans `fich` une suite de `nbre` octets dont le début est pointé par `ptr`.
- Constante `EOF` qui est renvoyée par certaines opérations sur les fichiers lorsque la fin du fichier est franchie.
- Exemples : `fgetc`, `fscanf`.

Règles de manipulation des fichiers

- Lors d'un `fopen` toujours tester que le résultat renvoyé n'est pas `NULL`.
  - ... et prévoir du code pour les cas où le fichier n'a pas pu être ouvert.
  - Une variable supplémentaire, `errno`, est instanciée par certaines fonctions (dont `fopen`) et donnant des détails sur la raison de l'impossibilité d'ouverture du fichier (voir pages du manuel !).
- Ne jamais oublier de fermer un fichier lorsqu'il n'est plus nécessaire.
- Ne jamais ouvrir le même fichier simultanément dans deux variables dont une est en écriture !
- Tester, lors de l'appel de `fread`, si on a pu lire le nombre d'octets qu'on voulait – cela permet d'ailleurs de détecter la fin du fichier !
- Tester, de manière plus générale, la valeur de retour de chaque fonction – cela permet de détecter des éventuelles erreurs de lecture/écriture sur disque, et de prévoir du code pour ces cas spéciaux.