TP C avancé n^o 4
G. Subrenat, Y. Bertrand, N. Courilleau, D. Meneveaux

Licence Informatique L3 - 2021-2022
UFR SFA - Université de Poitiers

Reprise en main

Table des matières

1 Résumé et cadre de travail

Le but de cette description est d'avoir une vue d'ensemble de la manipulation des fichiers en C, et notamment :

- la notion d'entrées/sorties bufferisées
- la différence entre mode binaire et mode texte

Une suite d'exercices permet d'appliquer tous ces concepts.

2 Mode bufferisé ou non

Point de cours

Il y a deux familles de fonctions pour manipuler les fichiers :

- open close lseek write read dprintf ...
- fopen fclose fseek fwrite fread fprintf fscanf fflush ...

Famille open :
Ce sont des appels système a priori moins portables (n'existent pas sous Windows je pense). Les entrées/sorties sont bas niveau, i.e. non bufferisées.

Famille fopen :
Ce sont des fonctions de la bibliothèque standard du C et donc plus portables. Les entrées/sorties sont plus haut niveau, i.e. bufferisées.

Des écritures bufferisées ne sont pas écrites directement sur disque mais stockées en mémoire. Ce sont les fonctions qui décident quand écrire sur disque (buffer plein, fermeture du fichier, ...). C'est bien plus efficace (moins d'accès disque et avec des volumes de données plus grands), mais on maîtrise moins ce que l'on fait.

Des écritures non bufferisées sont immédiatement écrites sur disque avec une inversion des avantages et inconvénients par rapport au mode bufferisé.

3 Mode binaire ou mode texte

Point de cours

Lorsqu'on écrite en mode "texte", le résultat est compréhensible par un humain, i.e. le fichier peut être ouvert avec un éditeur de texte.

En mode binaire, on écrit les données selon le codage de la machine et il faut un logiciel dédié pour lire le fichier (style *od*).

Avantages du mode texte :
- lisible par l'humain
- portable d'une architecture matérielle à une autre

Inconvénients du mode texte :
- nécessite des conversions de codage
- la place occupée par les données peut dépendre des valeurs de ces données
- relecture des données par programme plus difficile (du fait de la taille variable)

Voyons par l'exemple avec l'écriture d'un int.
En mode texte (avec fprintf ou dprintf) :
- il est converti en suite de caractères qui sont mis les uns à la suite des autres dans le fichier (123 sera transformé en '1' '2' et '3')
- le nombre de caractères écrits va dépendre de la valeur (3 caractères pour 123, 7 caractères

pour -123456, ...)
En mode binaire (avec fwrite ou write) ;
- c'est directement l'image mémoire du nombre qui est écrite
- la taille est fixe quelle que soit la valeur (4 pour un int, 8 pour un long [a])

Et pour la lecture
En mode texte :
- il faut faire la conversion inverse
- il faut savoir combien de caractères lire
- le fichier peut être lu sur d'autre architectures
En mode binaire
- recopie des valeurs du fichier directement en mémoire
- on sait combien d'octets il faut lire

Ecriture de deux nombres
En mode texte
- il faut les séparer par un séparateur (sic) : espace ou autre en effet si on écrit 12 et 34 sans précaution, dans le fichier il y aura "1234" et impossible lors de la lecture de savoir quoi lire (1 et 234 ? 12 et 34 ? 123 et 4 ?)
En mode binaire
- il n'y a pas de soucis, pour les int les 4 premiers octets contiennent le premier nombre, les 4 suivants le second

Lecture d'un tableau de nombres, on veut récupérer le 13me
En mode texte
- il faut lire les nombres un par un en sautant les espaces jusqu'à arriver au 13me
En mode binaire
- on se déplace (pour les int) à la postion 48 (4*(13-1))
et on lit 4 octets

Portabilités inter-architecture
Prenons l'exemple des int : selon les architectures l'ordre des 4 octets est en big endian ou little endian.
Big endian : les octets sont ordonnés du poids fort vers le poids faible
Little endian : c'est le contraire
Exemple : 3270897 (en hexa : 0031E8F1)
- les valeurs des quatre octets sont : 0 49 232 241
- en big endian, les octets sont dans cet ordre en mémoire
- en little endian, l'ordre est 241 232 49 0
Si on écrit le fichier sur une architecture et qu'on le lit sur une qui est inversée, c'est la catastrophe. Si on ecrit 3270897 en little endian, et qu'on le lit en big endian, on obtiendra -236441344.
(cf. fonctions *htonl*, *htons*, *ntohl*, *ntohs* si vous êtes intéressés)

a. Nous ferons cette hypothèse dans le document, mais la taille d'un *int* peut varier d'une architecture à l'autre.

4 Liste de quelques fonctions

Point de cours

Haut-niveau (bufferisé) :

- fopen : ouverture
- fclose : fermeture
- fwrite : écriture en binaire
- fread : lecture en binaire
- fprintf : écriture en mode texte
- fscanf : lecture en mode texte

- fseek : se déplacer dans le fichier
- fflush : forcer l'écriture du buffer sur disque

Bas-niveau (non bufferisé) :
- open : ouverture
- close : fermeture
- write : écriture en binaire
- read : lecture en binaire
- dprintf : écriture en mode texte
- lseek : se déplacer dans le fichier

5 Exercices

Il s'agit de manipuler les entrées-sorties sur les fichiers, et de bien différencier les modes binaire et texte.

Les bibliothèques ci-dessous sont généralement nécessaires pour tous les exercices.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
```

Il pourra être nécessaire d'en ajouter d'autres.

Les squelettes des fichiers sources sont fournis.

5.1 Exercice A : binaire vs. texte

Cf. fichier *exerciceA.c*

Ouverture et fermeture de fichiers : le but est d'effectuer des manipulations de base :
- Avec *open*, *dprintf* et *close*, créer le fichier *exoA_1_test* et écrire l'entier 12 dedans.
- Avec *open*, *write* et *close*, créer le fichier *exoA_2_test* et écrire l'entier 12 dedans.
- Avec *fopen*, *fprintf* et *fclose*, créer le fichier *exoA_3_test* et écrire l'entier 12 dedans.
- Avec *fopen*, *fwrite* et *fclose*, créer le fichier *exoA_4_test* et écrire l'entier 12 dedans.

Examinez les 4 fichiers générés :
- avec un éditeur de texte
- avec la commande : *od -Ad -w10 -t d1z <nom fichier>*
- avec la commande : *od -Ad -w10 -t x1z <nom fichier>*
- avec la commande : *hexdump -C <nom fichier>*
- avec la commande : *hd <nom fichier>*

Comprenez et expliquer la signification de chaque octet.

5.2 Exercice B : curiosité binaire

Cf. fichier *exerciceB.c*

Créez le fichier *exoB_1_test* et écrivez dedans en binaire (avec *fwrite*) le nombre n1 = 1952540739.

De la même façon, créez le fichier exoB_2_test avec le nombre n2 = 1130914164.

Ouvrez les fichiers générés avec un éditeur de texte et expliquez.

5.3 Exercice C : écriture/lecture d'une chaîne de caractères

Cf. fichier *exerciceC.c*

Écrire une fonction ;
*void ecritChaine(const char *nomFichier, const char *s)*
qui sauvegarde une chaîne de caractères dans un fichier.

Écrire une fonction :
*void litChaine(const char *nomFichier, char *s)*
qui lit une chaîne de caractères dans un fichier (chaîne écrite par la fonction précédente). Il ne faut pas effectuer la lecture caractère par caractère, mais lire la chaîne d'un bloc, autrement dit en une seule instruction (il n'est pas interdit de stocker des informations supplémentaires dans le fichier lors de l'écriture).
Note : on stocke le résultat dans s qui doit être suffisamment grande. Il est de la responsabilité de l'appelant de s'en assurer ; la fonction ne doit pas (et ne peut pas d'ailleurs) le faire.

Dans la fonction *main*, faites le code qui écrit une chaîne stockée dans une variable dans un fichier et qui relit cette même chaîne dans une autre variable, puis qui compare les valeurs de ces deux variables.

5.4 Exercice D : string input/output (bis repetita)

Cf. fichier *exerciceD.c*

On reprend le même problème que l'exercice précédent, mais on écrit et lit un tableau de chaînes de caractères.

Écrire une fonction ;
*void ecritTabChaines(const char *nomFichier, char * s[], int size)*
qui sauvegarde un tableau de chaînes de caractères dans un fichier.

Écrire une fonction ;
*void litTabChaines(const char *nomFichier, char * tab[], int *size)*
qui lit un tableau de chaînes de caractères dans un fichier (tableau écrit par la fonction précédente).

Comment savoir le nombre de chaînes à lire ?

Comment faire pour minimiser le nombre d'instructions de lecture : chaque chaîne doit être lue en une seule instruction (il n'est pas interdit de stocker des informations supplémentaires dans le fichier lors de l'écriture) ? ?

Notes :
- le tableau tab doit être suffisamment grand : c'est de la responsabilité de l'appelant de le pré-allouer ;
- *size* contiendra le nombre de chaînes lues.

Dans la fonction *main*, faites le code qui écrit un tableau stocké dans une variable dans un fichier et qui relit ce même tableau dans une autre variable, puis qui compare les valeurs de ces deux variables.
Le code pour gérer les tableaux est fourni.

5.5 Exercice E : tableaux d'entiers en binaire

Le but ici est d'écrire un tableau d'entiers et de ne relire qu'un seul élément, le tout en mode binaire avec les fonctions *fread* et *fwrite*.

Cf. fichier *exerciceE.c*

Écrire une fonction ;
*void ecritTabIntBin(const char *nomFichier, const int tab[], int n)*
qui écrit un tableau de *int* dans un fichier ; on ne met dans le fichier que les nombres du tableau et aucune autre information. L'écriture de tous les nombre doit être faite en une seule instruction.

Écrire une fonction ;
*int litIntPosBin(const char *nomFichier, int pos)*
qui lit un seul nombre (désigné par sa position) dans un fichier écrit par la fonction précédente.

Notes :
- Si l'indice est trop grand, on renvoie -1. On suppose qu'il n'y que des nombres positifs dans le fichier.
- On suppose aussi que la position passée en paramètre est positive.
- Pensez aux fonctions *ftell* et *fseek*.

Écrire une fonction ;
*bool remplaceIntPosBin(const char *nomFichier, int pos, int newVal)*
qui remplace un nombre (désigné par sa position) par un autre dans un fichier écrit par la fonction précédente. Si la position est trop grande, on ne fait rien et on renvoie false.

Dans la fonction *main*, faites le code qui teste les fonctions précédentes.

5.6 Exercice F : tableaux d'entiers en mode texte

Il s'agit donc du même exercice que le précédent, mais en mode texte, avec les fonctions *fprintf(..., "%d", ...)* et *fscanf(..., "%d", ...)*.

Cf. fichier *exerciceF.c*

Écrire une fonction ;
*void ecritTabIntAsc(const char *nomFichier, const int tab[], int n)*
qui écrit un tableau de *int* dans un fichier ; on ne met dans le fichier que les nombres du tableau et aucune autre information : les nombres seront séparés par un espace.

Écrire une fonction ;
*int litIntPosAsc(const char *nomFichier, int pos)*
qui lit un seul nombre (désigné par sa position) dans un fichier écrit par la fonction précédente.
Notes :
- Si l'indice est trop grand, on renvoie -1. On suppose qu'il n'y que des nombres positifs dans le fichier.
- On suppose aussi que la position passée en paramètre est positive.

Écrire une fonction ;
*bool remplaceIntPosAsc(const char *nomFichier, int pos, int newVal)*
qui remplace un nombre (désigné par sa position) par un autre dans un fichier écrit par la fonction précédente. Si la position est trop grande, on ne fait rien et on renvoie false. Attention c'est particulièrement pénible à faire ; ne faites cette fonction que lorsque le reste du TP est terminé.

Dans la fonction *main*, faites le code qui teste les fonctions précédentes.

5.7 Exercice G : Lecture / écriture d'une structure

Cf. fichier *exerciceG.c*

Soit le type suivant :

```
1 struct S
2 {
3     int i;
4     char c;
5     float f;
6     int tab[10];
7 };
```

Écrire une fonction ;
*void ecritStructBin(const char *nomFichier, const struct S *s)*
qui écrit, en binaire, une structure dans un fichier ; l'écriture doit être faite en une seule instruction.

Écrire une fonction ;
*void litStructBin(const char *nomFichier, struct S *s)*

qui lit une structure dans un fichier créé par la fonction précédente ; la lecture doit être faite en une seule instruction.

Écrire une fonction ;
void tailleStruct()
qui affiche la taille totale de la structure, puis la somme des tailles des 4 champs.

Dans la fonction *main*, faites le code qui test les fonctions précédentes.

Lorsqu'on compare la taille de la structure, la somme des tailles des 4 champs et la taille du fichier, a-t-on les mêmes nombres ?
Si ce n'est pas le cas, quelles sont les données en trop ou en moins [1] ?

5.8 Exercice H : écriture non bufferisée

Cf. fichier *exerciceH.c*

Avec *open*, *dprintf* et *close* faire le programme suivant :
- créer le fichier *exoH_test*
- écrire dedans la chaine "AA\n"
- écrire dedans la chaine "BB\n"
- écrire dedans la chaine "CC\n"
- attendre 10 secondes (fonction *sleep*)
- écrire dedans la chaine "DD\n"
- fermer le fichier

Le but est d'interrompre violemment le programme avec Ctrl-C lors de l'attente et de regarder le contenu du fichier partiellement rempli.

Est-ce le résultat attendu ?

5.9 Exercice I : écriture bufferisée

Cf. fichier *exerciceI.c*

Il s'agit exactement du même exercice mais avec les fonctions *fopen*, *fprintf* et *fclose* et le fichier *exoI_test*.

A-t-on le même comportement que l'exercice précédent ? Comment l'expliquez-vous ? (c'est une question d'entrées-sorties bufferisées [2])

1. Il faut espérer que l'on ne perd pas de données !
2. Regardez aussi la fonction *fflush*