I2181A : Langage C - Modularisation (TP4)

Les fichiers (binaires)

Reprenez le programme `biblio` écrit lors du TP 2 (solution fournie sur Moodle). Ajoutez deux fonctions au module *biblio* :

- `ecrireFichier()` : cette fonction sauve des livres dans un fichier. Elle reçoit un fichier binaire ouvert en écriture dans lequel elle enregistre un tableau de livres.
- `lireFichier()` : cette fonction charge les livres d'un fichier dans un tableau. Elle reçoit un fichier binaire ouvert en lecture, lit et copie son contenu dans un tableau de livres. Nous vous conseillons de placer un livre dans la table en faisant appel à la fonction `ajouterLivre()` (la taille de la table est gérée dynamiquement en fonction du nombre de livres lus).

1. Programme de test

Dans un premier temps, écrivez deux programmes.

- Le premier programme lit des livres au clavier et les sauve dans un fichier grâce à la fonction `ecrireFichier`. Le nom du fichier généré est donné sur la ligne de commande. Comme le faisait le programme *testBiblio* du TP2, tant que la fin de fichier n'est pas atteinte sur l'entrée standard :
  a) lire les informations d'un livre, à raison d'une information par ligne, une ligne vide séparant deux livres
  b) copier les informations dans une structure Livre
  c) placer le livre dans la table grâce à la fonction *ajouterLivre()* (la taille de la table doit être gérée dynamiquement en fonction du nombre de lignes lues)
  d) une fois tous les livres lus au clavier, sauvegarder la bibliothèque dans un fichier binaire à l'aide de la fonction `ecrireFichier`.
- Le second programme lit les livres d'un fichier grâce à la fonction `lireFichier` et les affiche à l'écran grâce à la fonction `afficherBib`. Le nom du fichier de données est fourni sur la ligne de commande.

Voici un exemple d'appel de ces deux programmes [1] :

```
./testEcriture biblio < bib.txt
./testLecture biblio
```

[1] Vous pouvez voir le contenu d'un fichier (binaire) à l'aide de la commande *xxd*, convertisseur hexadécimal de Linux : `xxd biblio`

2. Application

Nous vous demandons d'écrire un programme qui permet de créer une bibliothèque et la mettre à jour (càd. lui ajouter des livres) à la demande.

L'usage du programme sera :

```
./biblioMaj [fichierIn] fichierOut
```

Si un seul nom de fichier est passé en argument,

```
./biblioMaj fichierOut
```

les enregistrements lus au clavier seront sauvegardés dans le fichier `fichierOut`.

Si deux noms sont présents sur la ligne de commande,

```
./biblioMaj fichierIn fichierOut
```

le premier (`fichierIn`) est celui du fichier qui contient déjà des enregistrements qu'il faudra lire et le second (`fichierOut`) celui du fichier où le programme sauvegardera l'ensemble des enregistrements (i.e. ceux de `fichierIn` et les nouveaux introduits au clavier) :

Testez votre programme à l'aide de la séquence suivante (voir fichier *TP4_exemple_d'exécution.txt*) :

`$ ./biblioMaj testIN < bib.txt`

(le fichier `testIN` est écrit avec la fonction *ecrireFichier*).

`$ ./biblioMaj testIN testOut < tintin.txt`

(le fichier `testIN` est lu avec la fonction *lireFichier* et le fichier `testOUT` est écrit avec *ecrireFichier*).

Le programme doit :

1. s'il a reçu deux arguments sur la ligne de commande : lire tous les enregistrements de `fichierIn` et les mettre en table grâce à la fonction `lireFichier()` du module *biblio*
2. afficher la table générée
3. trier la table grâce à la fonction `qsort`, la clé de tri primaire sera l'année d'édition, les clés secondaires seront l'éditeur puis l'auteur (inspirez-vous de la solution du TP2 fournie)
4. afficher la table triée
5. grâce à la fonction `ecrireFichier()` du module *biblio*, écrire tous les éléments de la table triée dans le fichier dont le nom (`fichierOut`) est passé en argument sur la ligne de commande ; si ce fichier n'existe pas, il est créé, sinon il est ouvert en écriture et écrasé.

On suppose que les données lues à l'entrée standard sont conformes au format attendu.

Codes d'erreurs dans le programme principal :

- Si le programme est appelé avec moins d'un argument ou plus de deux, il affiche l'usage sur la sortie d'erreur et s'arrête avec un code d'erreur égal à 1.
- Si `fichierIn` n'existe pas ou n'est pas ouvrable en lecture, un message d'erreur explicite est affiché à la sortie d'erreur et le programme se termine avec un code d'erreur égal à 10.
- Si `fichierOut` ne peut être ouvert en écriture, un message d'erreur spécifique est affiché à la sortie d'erreur et le programme se termine avec un code d'erreur égal à 11.
- Si le programme rencontre des problèmes pour lire ou pour écrire dans un fichier, l'utilisateur doit être averti par un message qui précise le type d'erreur avant que le programme ne se termine prématurément avec un code d'erreur à 12 (pour un problème en lecture) ou 13 (pour un problème en écriture).
- Les éventuels problèmes d'allocation de la mémoire doivent aussi être signalés et provoquer l'arrêt du programme avec un code d'erreur à 20.

Pour rappel, la commande Linux permettant d'afficher le code d'erreur renvoyé par le dernier programme exécuté dans le terminal est : `echo $?`

Bonus

Réécrivez votre fonction `lireFichier()`, sans que celle-ci appelle `ajouterLivre()`.

`lireFichier()` devra donc gérer elle-même l'allocation dynamique de la bibliothèque. Cela permet de lire le fichier par bloc de NB_LIVRES livres directement dans bib. Cela est beaucoup plus efficace que de passer par un buffer *Livre* et d'ajouter les livres un par un dans la bibliothèque.