Introduction C - TP 4
Application de gestion d'une bibliothèque

Marc Hartley

Février 2025

1 Introduction

Ce TP est conçu pour vous permettre de mettre en pratique les compétences acquises lors de vos cours de programmation en C, en développant un système complet de gestion de bibliothèque. Ce projet simule des aspects réels de gestion de données complexes, une compétence essentielle pour tout développeur logiciel.

Objectifs du TP L'objectif principal de ce TP est de développer un système de gestion de bibliothèque en mémoire vive, sans persistance sur disque pour simplifier les interactions et se concentrer sur la structure du programme et la manipulation des données. Vous allez créer et manipuler des structures pour les livres, les clients et les emprunts, et mettre en œuvre des fonctionnalités de base telles que l'ajout, la suppression, la modification et la recherche d'informations dans une base de données de livres, de clients et d'emprunts de livres.

Quelques conseils pour y arriver Ce TP concentre plusieurs heures de travail, bien plus qu'une simple séance de TP. Voyez cela comme un projet court. De ce fait, la qualité de votre code joue un rôle important dans la réussite du projet, voilà quelques points que vous devez garder à l'esprit :

- Commencez par concevoir clairement vos structures de données avant de coder les fonctions.
- Testez chaque fonction individuellement avant de les intégrer.
- Utilisez des commentaires pour améliorer la lisibilité de votre code.

2 Présentation du problème

Bienvenue dans ce projet commandé par "LibrairIG", une librairie qui, face à l'augmentation de sa clientèle et la diversification de son stock, a décidé de remplacer ses méthodes manuscrites par un système moderne de gestion. LibrairIG vous a embauchés pour développer un logiciel interne qui optimisera et automatisera la gestion des informations des clients, l'inventaire des livres et les enregistrements des emprunts. Ce projet est une excellente opportunité pour vous de mettre en pratique vos compétences en programmation en C tout en contribuant à la transformation numérique d'une entreprise locale, améliorant ainsi son efficacité opérationnelle et la qualité de son service client.

Le système de gestion de bibliothèque que vous allez développer comprendra trois composants principaux :

- Gestion des livres : Vous allez concevoir des fonctions pour ajouter, supprimer, modifier et afficher des informations sur les livres.
- Gestion des clients : Vous implémenterez des fonctionnalités pour gérer les informations des clients, telles que l'ajout de nouveaux clients, la mise à jour de leurs informations, et l'affichage de leurs détails.

- Gestion des emprunts : Vous créerez des fonctions pour enregistrer les emprunts de livres par les clients, afficher les détails des emprunts, et gérer les retours.

3 Créer les fonctionnalités "core"

3.1 Les "Book"

Pour débuter la création du système de gestion pour la bibliothèque LibrairIG, nous allons structurer et coder les fonctionnalités liées à la gestion des livres.

Créez les fichiers "Book.h" et "Book.c".

Créez une structure `Book` qui contiendra toutes les informations nécessaires sur un livre. La structure devra inclure :

- isbn (chaîne de caractères) : pour stocker le numéro ISBN du livre. Dans notre "base de données", il servira d'identifiant unique.
- title (chaîne de caractères) : pour le titre du livre.
- author (chaîne de caractères) : pour le nom de l'auteur.
- year (entier) : l'année de publication du livre.

Écrivez une fonction printBook qui prend un pointeur vers une structure Book et affiche ses informations de manière formatée. Cette fonction sera utile pour vérifier les entrées et pour les opérations de débogage.

```
void printBook(const Book *book) {
    // ...
}
```

On va aussi créer les fonctions pour créer et modifier un Book :

```
Book* createBook(char* isbn, char* title, char* author, int year) {
    // ...
}

Book* modifyBook(Book* bookToModify, char* isbn, char* title, char* author, int year) {
    // ...
    return bookToModify;
}
```

Compilation : Pour des raisons incertaines, un "warning" peu utile peut se présenter à vous. Vous pouvez ajouter un argument dans la ligne de compilation :

```
gcc -Wall -Wno-stringop-overflow main.c Book.c -o mon_programme
```

Question :

- Dans le futur, on voudra probablement retirer certains livres de notre bibliothèques (trop abîmés, trop vieux, trop nuls, etc...). Comment gérer la suppression de livres, sachant que les emprunts seront liés à ces livres ?
- Connaissant cette même contrainte, est-ce judicieux de pouvoir modifier le champ "isbn" d'un livre ?

ISBN	Titre	Auteur	Année de publication
978-0451524935	1984	George Orwell	1984
978-0553293357	Foundation	Isaac Asimov	1951
978-1451673319	Fahrenheit 451	Ray Bradbury	1953

Enfin, testons nos fonctions. Dans la fonction `main()` du fichier "main.c", créez les livres et affichez ces livres :

Attention, le tableau précédent a une erreur bête concernant l'année de publication du livre "1984". Utilisez la fonction "modifyBook" pour corriger cela et ré-affichez-le.

ISBN	Titre	Auteur	Année de publication
978-0451524935	1984	George Orwell	1949

Attention : Verifiez bien que la compilation fonctionne. Vous avez besoin de "linker" (lier) les codes "main.c" et "Book.c" dans la commande GCC. Pour éviter les erreurs de "link", considérez que "main.c a besoin des fonctions de Book.c" donc la commande doit être "gcc -Wall main.c Book.c -o prog_biblio" (avec "main.c" AVANT "book.c").

Tout de bon ? Parfait !

3.2 La "Library"

Pour l'instant on a des livres "sur feuilles volantes". On va continuer à structurer notre code proprement en stockant tout dans une structure supplémentaire, qui peut être vue comme une "surcouche".

Créez les fichiers "Library.h" et "Library.c". Ils doivent contenir la structure "Library". Pour l'instant elle contient seulement une liste (tableau) de Books ainsi que la taille de ce tableau. On va bientôt ajouter bien plus d'attributs !

Question : On a besoin d'utiliser notre structure "Book", créée dans un autre fichier ("Book.h"). Comment peut-on avoir accès à cette structure dans "Library.h" ?

Attention : Verifiez bien que la compilation fonctionne. Vous avez besoin de "linker" (lier) les codes "main.c", "Book.c", et "Library.c" dans la commande GCC. Pour éviter les erreurs de "link", considérez que "main.c a besoin des fonctions de Library.c, qui lui-même a besoin des fonctions et structures de Book.c" donc la commande doit être "gcc -Wall main.c Library.c Book.c -o prog_biblio" (avec "main.c" PUIS "Library.c" PUIS "book.c").

Maintenant, on peut rentrer dans le principe de "surcouche", c'est-à-dire créer des fonctions très semblable à ce que l'on a déjà fait, mais qui ajoutent une *couche de sécurité* :

Créez la fonction `addBookToLib()`. Si la bibliothèque contient déjà un livre avec le même *identifiant unique* "isbn", affichez un message d'erreur. Sinon, le nouveau Book* est ajoutée à la liste de livres de la bibliothèque (et incrémente (ajoute 1 à) la taille de la liste de livres). Cette fonction doit utiliser la fonction "createBook" de "Book.c".

Créez une fonction `getBookFromLib()` qui, étant donné un "ISBN", retourne le `Book*` associé (ou "NULL" si aucun livre n'existe avec cet ISBN).

Créez la fonction `displayAllBooksInLib()` qui affiche tous les livres dans la liste sous la forme "titre, auteur (année)".

Créez la fonction `modifyBookFromLib()` qui, étant donné un "ISBN", peut modifier les informations d'un livre. Vous aurez besoin de `getBookFromLib()` et `modifyBook()`.

De la même façon, créez `removeBookFromLib()`. Souvenez-vous de votre réponse à la question "Comment gérer la suppression de livres ?". Devez-vous modifier la taille de la liste de livres ?

3.3 Les "Customer"

Pour poursuivre le développement de notre système de gestion pour la bibliothèque LibrairIG, nous allons structurer et coder les fonctionnalités liées à la gestion des clients.

Créez les fichiers "Customer.h" et "Customer.c".

Définissez une structure `Customer` qui contiendra toutes les informations nécessaires sur un client. La structure devra inclure :

- customerID (entier) : pour stocker l'identifiant unique du client.
- name (chaîne de caractères) : pour le nom complet du client.
- address (chaîne de caractères) : pour l'adresse du client.

Écrivez une fonction printCustomer qui prend un pointeur vers une structure Customer et affiche ses informations de manière formatée. Cette fonction sera utile pour vérifier les entrées et pour les opérations de débogage.

```
void printCustomer(const Customer *customer) {
    // ...
}
```

On va aussi créer les fonctions pour créer et modifier un Customer :

```
Customer* createCustomer(int customerID, char* name, char* address) {
    // ...
}

Customer* modifyCustomer(Customer* customerToModify, int customerID, char* name, char* address) {
    // ...
    return customerToModify;
}
```

Question :

- Comment gérer les situations où un client déménage ou change de nom ? Quelles sont les implications sur la gestion des emprunts en cours ?
- Est-ce judicieux de permettre la modification du champ "customerID" d'un client ?
- Existe-t-il des types de données plus adaptées pour stocker "customerID" ?

Enfin, testons nos fonctions. Dans la fonction `main()` du fichier "main.c", créez des clients et affichez ces clients :

ID Client	Nom Complet	Adresse
C001	Alice Dupont	123 Rue de la Liberté
C002	Bob Martin	456 Avenue de la République
C003	Charlie Vincent	789 Boulevard des Étoiles

Utilisez la fonction "modifyCustomer" pour mettre à jour l'adresse d'Alice Dupont suite à son déménagement, et ré-affichez ses informations modifiées.

Si tout fonctionne, on peut passer à la suite !

ID Client	Nom Complet	Adresse
C001	Alice Dupont	987 Rue de la Paix

3.4 La "Library" #2

Maintenant, vous devez avoir une idée de ce qu'il y a à faire dans les fichiers "Library.h" et "Library.c" :

- En plus de la liste de livres (et sa taille), on veut stocker la liste des clients (et sa taille).
- Ajoutez toutes les fonctions de surcouche pour les clients : addClientToLib(), getClientInLib(), displayAllClients-FromLib(), modifyClientFromLib() et removeClientFromLib().

Pensez toujours à tester que toutes vos fonctions... fonctionnent.

3.5 Les "Borrow"

On est à deux doigts de faire tourner une bibliothèque. Il manque quand même le point essentiel d'une bibliothèque : l'emprunt de livres par des clients.

Vous avez eu des cours de SQL, donc vous savez comment gérer les relations entre plusieurs "tables". Prenez un petit moment pour réfléchir à la relation "Borrow" ("emprunter") entre les "Customer" et "Book" :

- Si uniquement 1 client peut emprunter uniquement 1 livre, on est dans une relation "One-to-One" (1-1). Hypothétiquement, comment modifieriez-vous les structures "Customer" et/ou "Book" pour les lier ?
- Si 1 client peut emprunter N livres, mais chaque livre ne peut être emprunté par qu'1 unique client, on est dans une relation "One-to-Many" (1-N). Hypothétiquement, comment modifieriez-vous les structures "Customer" et/ou "Book" pour les lier ?
- Si 1 livre peut être emprunté par N clients, mais chaque client ne peut emprunter qu'1 unique livre, on est dans une relation "One-to-Many" aussi (1-N). Hypothétiquement, comment modifieriez-vous les structures "Customer" et/ou "Book" pour les lier ?

Bien sûr dans notre cas, chaque client peut emprunter autant de livres qu'il souhaite, et les livres peuvent être emprunté plusieurs fois par des personnes différentes (à des dates différentes). Donc on est dans une relation de Many-to-Many (N-N). Cela signifie qu'on doit créer une table supplémentaire pour stocker toutes les transactions.

Dans les fichiers "Emprunt.h" et "Emprunt.c", créez la structure "Emprunt". Elle doit contenir :

- L'identifiant unique d'un client,
- L'identifiant unique d'un livre,
- Une date d'emprunt...

Wow wow wow ! On a pas de type de données pour les dates ! Vous le voyez venir... une nouvelle structure "Date" ! Celle là doit contenir :

- l'année (entier)
- le mois (entier)
- le jour (entier)

Ok, maintenant on peut créer, encore une fois, les fonctions "createBorrow()", "displayBorrow()", "modifyBorrow()" et "removeBorrow()".

Testez les fonctions pour être sûr qu'elles fonctionnent.

Vous pouvez utiliser ces données :

ID Client	ISBN	Date d'emprunt
C001	978-0451524935	2024/3/10
C002	978-0553293357	2025/1/25
C002	978-0451524935	2025/1/25

3.6 La "Library" #3

Grande surprise, on va "agrémenter" notre code de "Library.h" et "Library.c" pour la gestion de la bibliothèque en stockant aussi les emprunt !

- En plus de la liste de livres et de clients, on veut stocker les emprunts (et le nombre de ces transactions).
- Ajoutez toutes les fonctions de surcouche pour les emprunts : addBorrowToLib(), getBorrowInLib(), displayAllBorrowFromLib(), modifyBorrowFromLib() et removeBorrowFromLib().

Mais maintenant, on a beaucoup plus de choses à faire parce que nos tables (ou "Entités") "Book" et "Customer" sont enfin liées ! On peut par exemple :

- Créer la fonction "displayCustomer()" qui prend en argument un customerID et affiche les informations du client ainsi que la liste des livres qu'il a emprunté.
- Créer la fonction "displayBook()" qui prend en argument un ISBN et affiche les données du livre et les différentes transactions passées.

3.7 Les "Borrow" #2

Le directeur de LibrarIG regarde votre programme et remarque que les clients volent ses livres. En effet, les livres sont empruntés, mais jamais rendus !

Ajoutez un champ "returnedDate" dans les "Borrow".

On va pas se mentir, c'est maintenant que tout se complique ! Accrochez-vous encore un peu...

On DOIT remplir le nouveau champ "returnedDate" de "Borrow", mais si le livre n'est pas encore rendu, quelle date doit-on renseigner ?

Je vous propose deux possibilités, une qui fonctionne mais elle est "sale", et une autre "propre" qui demande plus d'efforts :

- Considérez que la date "0/0/0" est la "date nulle" (i.e. le livre n'est pas rendu). Pour verifier si la date est "valide", on peut utiliser "if (date.year != 0 && date.month == 0 && date.day == 0)".
- Ajoutez un champ "isValid" dans la struct "Date". Créez la fonction "Date initialize_date(int year, int month, int day)" pour créer des "Date" qui remplissent une date (et applique "isValid = true"), une fonction "void invalidateDate(Date* date)" qui passe le champ "isValid = false", et enfin une fonction "bool dateIsValid(Date date)" qui verifie si "date.isValid == true".

Voilà, maintenant on peut savoir si un livre a été rendu ou non. Dans "Library.c", ajoutez la fonction "returnBookToLib()" qui prend en argument un ISBN et une date et qui modifie l'entrée "Borrow" associé à cet ISBN et encore non-rendu pour modifier le champ "returnedDate".

3.8 Retards

Le directeur précise que normalement, un livre doit être rendu dans les 60 jours après un emprunt, sinon l'emprunt est "en retard".

On va considérer que tous mois contiennent 28 jours, histoire d'avoir un calendrier parfait, mais qu'il n'y a que 12 mois (par habitude)

Créez les fonctions "arithmétiques" pour les dates :

- "Date normalizeDate(Date date)" doit "normaliser"/"réguler" une date. Par exemple, si "date = 2025/10/34", cette fonction retourne "2025/11/6" (trop de jours dans un mois). Pour "date = 2020/12/34", cette fonction retourne "2021/1/6" (trop de mois).
- "addDate(Date date1, Date date2)" doit "additionner" deux date. Exemple "2020/10/3" + "0/0/5" = "2020/10/8" ou encore "2020/10/20" + "0/0/60" = "2020/12/24"
- "compareDate(Date date1, Date date2)" doit retourner 0 si les deux dates sont identiques, -1 si date1 est antérieur à date2 et 1 si date1 est posterieur à date2.
- "char* dateToString(Date date)" qui retourne justement chaque date sous la forme d'une chaine de caractère de la forme "year/month/day" dans le cas "sale" (un format "%d/%d/%d") ou "year/month/day/isValid" dans le cas "propre" (un format "%d%d%d%d" avec "isValid" affiché comme "0" ou "1").

Maintenant, créez la fonction "isBorrowLate()" qui retourne true si l'emprunt est "en retard" et false sinon.
Voilà quelques données pour tester :

ID Client	ISBN	Date d'emprunt	Date de rendu
C001	978-0451524935	2024/3/10	2024/8/13 (retard)
C002	978-0553293357	2025/1/25	Non rendu
C002	978-0451524935	2025/1/25	2025/1/28

C'était facile, non ?

4 Gestion de fichiers

Vous avez probablement remarqué que jusque là on a travaillé uniquement dans la mémoire vive, c'est-à-dire que tout recommence à zéro quand on relance le programme. Pas fou pour la multi-nationale LibrairIG. Pourquoi pas garder la mémoire sur le disque dur ? Pour ça, on va jouer avec les fichiers !

4.1 Stocker les données

On va partir de l'état actuel de notre programme. Pour l'instant, à chaque execution, la liste de livres, de clients et d'emprunts se remplit. Modifions les fonctions pour que tout s'enregistre sur le disque.

Créez une fonction "saveAllData()" dans "Library.h/c". Voilà le pseudo-code de cette fonction (gardez un espace avant et après les ";") :

```
DEBUT saveAllData(nomFichierBooks, nomFichierCustomers, nomFichierBorrows) :
    "Ouvrir" fichier "nomFichierBooks"
    POUR i allant de 0 à nombreDeLivres :
        Ecrire ISBN ; Titre ; Auteur ; Annee ; estRetiré (0 ou 1) \n
    FIN POUR
    "Fermer" fichier

    Ouvrir fichier "nomFichierCustomers"
    POUR i allant de 0 à nombreDeClients :
        Ecrire ID client ; Nom ; Adresse \n
    FIN POUR
    Fermer fichier

    Ouvrir fichier "nomFichierBorrows"
    POUR i allant de 0 à nombreDEmprunts :
        Ecrire ISBN ; ID client ; Date emprunt ; Date retour \n
    FIN POUR
    Fermet fichier
FIN
```

Questions :

- Quel doit être le mode d'ouverture ? Pourquoi ?

Verifiez que les données sont correctes, qu'aucune donnée n'a été perdue dans le processus.

4.2 Charger les données

Super, on a réussi à enregistrer nos données. Maintenant on peut essayer de "charger" des données.

Conseil : Créez une copie des fichiers de sauvegarde. Avec une mauvaise manipulation de lecture vous pouvez inintentionellement vider ces fichiers, donc gardez une "backup".

Dans un premier temps, on va simplement essayer de lire un fichier "nomFichierBooks". Pour rappel, chaque livre est stockée comme une chaine de caractères (une ligne). Donc un fichier contient une liste de chaine de caractères (pour information, une liste de char* = un tableau de char* = un char**).

Dans "Library.h/c", créez la fonction "initializeLibrary()" qui va permettre de lire les fichiers de sauvegarde. Dans "main()", mettez en commentaire toutes les instructions qui vous ont permis de créer des livres, des clients et des emprunts à la main.

"initializeLibrary()" doit elle-même appeler les fonctions "lireBooks()", "lireCustomers()" et "lireBorrows()". Ces trois fonctions sont très similiaires !

Voilà un premier pseudo-code pour analyser le fichier :

```
DEBUT lireBooks(nomFichierBooks) :
    Ouvrir fichier "nomFichierBooks"
    "Reserver" une chaine de caractère "buffer" de 1000 caractères de long
    TANT QUE fichier n'a pas atteint le End Of File (EOF) :
        Lire une ligne du fichier et la stocker dans "buffer"
        >> Analyser le contenu de "buffer" pour créer un objet "Book" <<
        Ajouter le Book dans la Library
```

```
    FIN TANT QUE
FIN
```

Mais qu'est-ce qui se passe dans la partie
`>> Analyser le contenu de "buffer" pour créer un objet "Book" <<` ?
On va devoir "parser" (analyser) chaque ligne. Dans un premier temps, affichez chaque "buffer" pour être sûr que le fichier est bien lu ligne par ligne.

Une solution est d'utiliser "sscanf" (une variante du "scanf" pour "String scanf") qui permet de parser du texte à partir d'une autre chaine de caractères-

```
char ISBN[20];
char title[100];
char author[100];
int year;

sscanf(buffer, "%s ; %s ; %s ; %d", ISBN, title, author, year);
```

Il n'y a plus qu'à faire la même chose pour les clients et les emprunts !

5 Interaction utilisateur

Notre programme prendrait beaucoup de valeur ajoutée avec une "bonne" interface utilisateur (on est en ligne de commande, donc calmos).

L'exercice ici est de manipuler les entrées utilisateur. Pour l'affichage, exploitez les fonctions créées dans les parties précédentes.

Créez un menu principal qui doit proposer de :

- 1. Afficher tous les livres
- 2. Afficher tous les clients
- 3. Afficher tous les emprunts
- 4. Créer un emprunt
- 5. Déclarer un retour
- 6. Ajouter/Modifier/Retirer un livre
- 7. Ajouter/Modifier/Supprimer un client
- 8. Modifier/Supprimer un emprunt

Si vous selectionnez 1. Afficher tous les livres :

- Affiche tous les livres (numérotés de 1 à nombreBooks)
- Tapez un numéro : affiche les données du livre, puis retourne au menu précédent
- Tapez "exit" pour revenir au menu précédent (ne soyez pas sensible à la casse)

- Tapez ": [du texte]" : affiche une sous-partie des livres qui contiennent "[du texte]" dans leurs données (ISBN, titre, auteur, année). Soyez "insensitif à la casse", et attention à la comparaison avec "year", qui est un entier.

Si vous selectionnez 2. Afficher tous les clients :

- Affiche tous les clients (numérotés de 1 à nombreCustomers)
- Tapez un numéro : affiche les données du client, puis retourne au menu précédent
- Tapez "exit" pour revenir au menu précédent (ne soyez pas sensible à la casse)
- Tapez ": [du texte]" : affiche une sous-partie des clients qui contiennent "[du texte]" dans leurs données (nom, prenom, adresse). Soyez "insensitif à la casse".

Si vous selectionnez 3. Afficher tous les emprunts :

- Affiche tous les emprunts

Si vous selectionnez 4. Créer un emprunt :

- Demande le ISBN
- Demande l'ID client
- Retourne au menu précédent

Si vous selectionnez 5. Déclarer un retour :

- Demande l'ID client
- Affiche les livres empruntés non-rendus
- Demande l'ISBN d'un livre
- Retourne au menu précédent

Si vous selectionnez 6. Ajouter/Modifier/Retirer un livre :

- Propose 3 options :
- A. Ajouter
- B. Modifier
- C. Retirer

Pour chaque option (6.A, 6.B, 6.C), le programme demande toutes les informations nécessaires sur un livre.
Si vous selectionnez 7. Ajouter/Modifier/Supprimer un client :

- Propose 3 options :
- A. Ajouter
- B. Modifier
- C. Supprimer

Pour chaque option (7.A, 7.B, 7.C), le programme demande toutes les informations nécessaires sur un client.
Si vous selectionnez 8. Modifier/Supprimer un emprunt :

- Propose 3 options :
- A. Modifier
- B. Retirer

Pour chaque option (8.A, 8.B), le programme demande toutes les informations nécessaires sur un emprunt.

5.1 Structurer un menu

Notre code ressemble à un gros spaghetti ! Voilà une idée d'implémentation de menus (non testé pour l'instant) :
Créez les fichiers "Menu.h" et "Menu.c".
Dans ce fichier, créez la structure "Menu". Un Menu contient une liste de proposition (tableau de char* = char**) et un tableau de pointeurs de fonction de type "void (*ma_fonction)(void)" [ici, le second void signifie "sans paramètre"].
Maintenant on peut avoir ce type de code :

```
Menu principal;
ajouterOption(&principal, "Afficher tous les livres", afficherMenuTousLivres);
ajouterOption(&principal, "Afficher tous les clients", afficherMenuTousClients);
ajouterOption(&principal, "Afficher tous les emprunts", afficherMenuTousEmprunts);
ajouterOption(&principal, "Quitter", quitter);

afficherMenu(principal);
```

Et ce code affichera :

```
1. Afficher tous les livres
2. Afficher tous les clients
3. Afficher tous les emprunts
4. Quitter
Votre choix > _
```

Conseil : Vous voyez que la liste des options d'un menu est "dynamique" (on ajoute des options à la volée). Soit vous gérez la taille des tableaux avec des mallocs et un peu d'huile de coude, soit vous réutilisez votre structure "Liste" créée dans un précédent TP.