PROJET DE PROGRAMMATION EN C

GESTIONNAIRE DE BIBLIOTHEQUE

SOMMAIRE

Présentation du cahier des charges et notice de l'application (page 3 et 4)

Présentation du cahier des charges à respecter tout au long du développement de ce projet, et de la notice de l'application.

L'architecture du projet (page 5)

Présentation de l'architecture du projet avec une description de chaque fichiers/dossiers.

Démonstration (page 6)

Quelques photos et explications du projet en action.

Explication (page 7-11)

Comment fonctionne le projet côté technique et pourquoi avoir procédé ainsi ?

- L'écran de connexion (page 6)
- Gestion des sauvegardes (page 7)
- Lecture et écriture des sauvegardes (page 7)
- Architecture de la fonction main (page 9)

Conclusion (page 12)

Ce que nous avons appris de ce projet, les difficultés rencontrées, ...

CAHIER DES CHARGES

Durant le développement de ce projet, nous avons du respecter certaines règles fixées par un cahier des charges. Les règles étants les suivantes :

- Les membres doivent êtres sauvegardés dans une base de données et comporter les informations suivantes :
  - Nom et prénom du membre
  - Adresse postale et mail du membre
  - Métier du membre
  - Liste des codes correspondant aux livres empruntés (3 emprunts max)

Pour respecter cette règle, nous avons décidé de passer par un fichier texte nommé ***membres.txt*** dans lequel se trouves les informations relatives aux membres

- Les livres doivent êtres sauvegardés dans une base de données et comporter les informations suivantes :
  - Titre et auteur du livre
  - Code servant à identifier le livre
    - Composé de 7 caractères, les 3 premiers servant à identifier le type de livre (ROM: Roman, BAD: Bande Dessinées,...) et 3 caractères pour le numéro du livre, les 2 séquences étant séparées par un tiret
  - Le nombre d'exemplaires du livre
  - Le nombre d'exemplaires disponibles

Comme pour les membres, nous sauvegardons ces données dans un fichier qui lui, est nommé ***livres.txt***

Enfin, comme pour tout produit, la conception d'un mode d'emploi se devait d'être faite.

LA NOTICE DE L'APPLICATION

Comment gérer un membre ?

- Un membre est géré par son identifiant créé automatique à l'ajout
- Un membre peut être ajouté seulement si l'adresse mail n'est pas existante
- Un membre peut réaliser au maximum trois prêts
- Un membre ne peut pas emprunter de livre s'il a un emprunt en retard

Comment gérer un livre ?

- Un livre peut être ajouté s'il n'est pas existant (c'est-à-dire s'il n'existe pas déjà un livre ayant le même titre et auteur dans le même genre)
- Le genre du livre doit correspondre à un des genres enregistrés
- Un livre ne peut plus être emprunté s'il n'y a plus d'exemplaires disponibles

Comment gérer un prêt ?

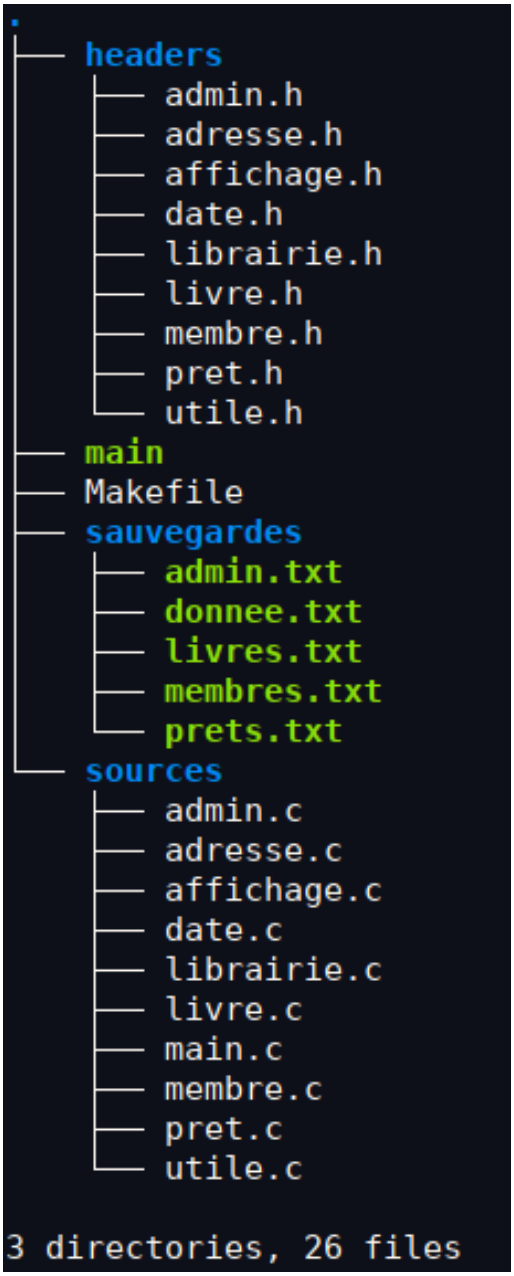
- Un prêt est géré par son numéro créé automatique à l'ajout
- Pour saisir un prêt il faut que toutes les caractéristiques saisies précédemment soit valides.
- La date de prêt et de retour maximum du livre ne sont pas à saisir. Ces valeurs sont calculées automatiquement.

Comment gérer les administrateurs ?

- Il est possible d'ajouter et de supprimer autant d'administrateurs désirés.
- Il n'est pas possible de supprimer l'administrateur de référence qui est celui du programmeur

Toute ces fonctionnalités rentrent dans le programme au niveau de la sécurisation des saisies.

ARCHITECTURE DU PROJET



headers :

Dossier contenant toutes les entêtes des fichiers C.

sources :

Dossier contenant toutes les sources C du projet, chaque fichier contient les fonctions relatives au nom du fichier (la fonction main du projet se trouve dans le fichier main.c).

sauvegardes :

Dossier contenant les différentes base de données vues dans la page précédente de ce rapport).

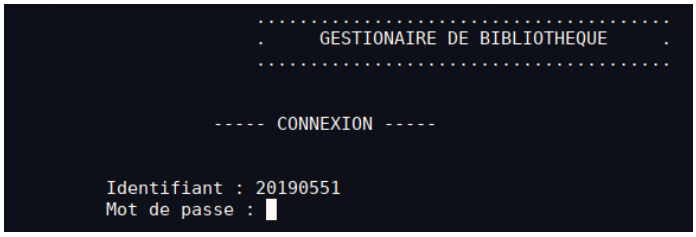
Makefile

Fichier servant à la commande ***make***, contenant différentes fonctions afin de simplifier le lancement du programme (***make run/all)***, la compilation du programme (***make)***, le nettoyage du projet (***make clean)*** ou bien le lancement du projet dans un débugger (***make debug)***. Ce fichier permet une homogénéité entre les 2 espaces de travail dans le groupe car en utilisant la commande ***make***, on s'assure que chaque personne compile le programme avec le même compilateur et les mêmes paramètres.

main

Exécutable du projet

DEMONSTRATION



À l'exécution de l'application, nous arrivons sur une page de connexion dans laquelle on doit entrer les identifiants (identifiants sauvegardés dans le fichier

Ce gestionnaire de connexion est géré par la fonction ***connexion*** située dans le fichier ***librairie.c*** et appelé dans la fonction ***main*** après initialisation de la librairie.

Cette fonction va boucler sur la demande des identifiants tant que ces derniers sont incorrects et ainsi bloquer la suite de l'exécution du programme tant que l'utilisateur n'entre pas d'identifiants corrects.

Une fois connecté, l'utilisateur arrive sur ce menu sur lequel il peut sélectionner l'action qu'il souhaite effectuer en entrant le numéro correspondant à cette dernière.

Une fois que l'utilisateur décide d'une action, ce choix sera comparé aux différents numéros dans la fonction main et la fonction correspondant au choix sera appelée, les fonctions étant toutes situées dans ***librairie.c*** et faisant appelles aux fonctions relatives à l'action demandée

exemple : l'utilisateur demande a supprimer un admin, la fonction ***supr_admin*** sera donc appelée et cette dernière effectuera les actions

EXPLICATION

Maintenant qu'on a vu l'aspect extérieur du projet, voyons ce qui se cache à l'intérieur.

L'écran de connexion :

À l'exécution du problème, ce dernière nous demande de se connecter. Pour se faire, nous avons créé une fonction ***void connexion(Admin **tab_admin, int *nb_identifiant)*** qui prend en premier paramètres un tableau contenant tous les administrateurs (représentés grâce a la structure :

```
typedef struct{
   int identifiant; //entier a 8 chiffres
   char mot_de_passe[30];
} Admin;
```

Chaque structure contenue dans le tableau sont générées grâce aux fonctions :

- void calcul_nb_admin(int *nb_identifiant);
- Admin **creer_tab_admin(int *nb_identifiant);

La fonction ***connexion*** ne retourne rien car elle bloque d'elle même la suite du programme, une fois cette fonction appelée, une boucle "do { ... } while (...)" qui, tant que les identifiants sont incorrect, va effectuer la fonction ***supr_console***, laquelle va lancer une commande dans le terminal (soit "cls" si le système est windows, sinon "clear"

```
void supr_console(void {
   system(strcmp(SYSTEME_EXPLOITATION, "win") == 0 ? "cls" : "clear");
}
```

Afin de connaitre le système, la macro-constrante *SYSTEME_EXPLOITATION* est utilisée, cette macro-constrante est définie dans le fichier ***utile.h*** à l'aide de condition de préprocesseur

```
#ifdef _WIN32
#define SYSTEME_EXPLOITATION "win" //windows
#elif __linux__
#define SYSTEME_EXPLOITATION "linux"
#elif __APPLE__
#define SYSTEME_EXPLOITATION "mac"
#else
#define SYSTEME_EXPLOITATION "unk" //systéme inconnu
#endif
```

La fonction pour vérifier la validité d'un administrateur reste relativement simple. En effet, elle va dans un premier temps demander un identifiant et un mot de passe puis boucler sur tous les admins passés dans le tableau afin de voir si cet identifiant est valide, et si c'est le cas, vérifier aussi le mot de passe. Si les 2 sont valides, alors la variable ***valide*** restera à TRUE et va donc retourner la valeur TRUE. Si l'identifiant ou le mot de passe est incorrect, alors la variable sera à FALSE et donc la fonction retournera FALSE.

Nous avons décidé d'utiliser les macro-constance TRUE et FALSE respectivement int 1 et int 0 afin de voir plus facilement ce que nous comparions dans les conditions.

Gestion des sauvegardes :

Afin de persister les données et après plusieurs recherches à ce sujet, plusieurs moyens de sauvegardes s'offraient à nous :

- Passer par une base de données SQL (MySQL, SQLite, ...)
- tout enregistrer dans des fichiers

L'option SQL bien que surement plus propre et professionnel ne nous convenait pas car trop compliqué à mettre en place. Nous avons donc décidé de passer par l'écriture et la lecture de fichier, de plus, on avait eu l'occasion d'utiliser cette technique lors du projet précédent. Pour le fichier, l'idée du JSON nous est parvenue cependant, par peur de manque de temps, nous avons préféré oublier cette idée car surement trop longue à mettre en place vu nos connaissances actuelles.

Au final, nous avons donc pris comme solution de passer par un fichier text pour chaque type de sauvegardes, c'est pour cette raison qu'il y a un ***admins.txt***, ***prets.txt***, ***livres.txt***,..

Lecture et écriture des sauvegardes

Les prêts, les membres, les livres, les admins et les données de la librairie ont chacun leur propre structure de donné.

Pour le fichier admins.txt, on savait que les informations stockées dedans serait un entier et une suite de caractères sans espaces, nous avons donc pu utiliser un simple fscanf et fprintf pour obtenir ses informations et écrire ses informations sur une ligne simple en suivant la forme "id : %d mp : %s\n".

Pour les autres fichiers, cette solution ne convenait pas à cause de la présence d'espaces. La solution a donc été de sauvegarder les différentes infos de la structure mais sur plusieurs lignes où chaque ligne correspond à un objet de la structure.

Par exemple :

```
Elle et lui
Marc Levy
ROM-001
nb_exemplaires : 2 - nb_exemplaires_dispos : 1
```

```
typedef struct {
    char titre[40];
    char auteur[40];
    char code[9]; //XXX-YYY\0
    int nb_exemplaires; //total
    int nb_exemplaires_dispo;
} Livre;
```

La première ligne va être écrite par un fprintf avec "%s\n" et mise dans l'objet titre, la seconde va être lue aussi par un fprintf avec "%s\n" et mise dans auteur... puis la dernière

Via un fprintf avec "nb_exemplaires : %d - nb_exemplaires_dispos : %d\n\n" afin de

```
{
    fprintf(fichier_livre, "%s\n", saisie->titre);
    fprintf(fichier_livre, "%s\n", saisie->auteur);
    fprintf(fichier_livre, "%s\n", saisie->code);
    fprintf(fichier_livre, "nb_exemplaires : %d - nb_exemplaires_dispos : %d\n\n", ...);
    fclose(fichier_livre);
}
```

marquer une démarcation entre chaque livre.

La lecture se fait de même mais avec des fscanf.

La relation entre les prêts et les membres se fait quant à elle ***Membre*** à l'aide du tableau ***int liste_emprunt[3];*** qui contient l'identifiant des différents prêts

Important : à chaque action (enregistrement ou suppression d'un membres, livre, prêt,...), le fichier correspondant est automatiquement édité afin de s'assurer que les changements sont biens appliqués même si le programme crash a la fin pour X raisons. De plus, cela évite de laisser en mémoire toutes les modifications que le programmes devrait faire à sa fermeture.

Les fonctions servant à l'enregistrement de fichiers sont :

- void ajout_admin_fichier_admin(FILE *fichier_admin, Admin *saisie)
- void ajout_livre_fichier_livre(FILE *fichier_livre, Livre *saisie)
- void ajout_membre_fichier_membre(FILE *fichier_membre, Membre *saisie)
- void ajout_pret_fichier_pret(FILE *fichier_pret, Pret *saisie)

Les fonctions servant à la lecture de fichiers sont :

- Admin **creer_tab_admin(int *nb_identifiant)
- void creer_tab_donnee(int tab[2], Donnee_livre *donne_livre)
- Livre **creer_tab_livre(int *nb_livre)
- Membre **creer_tab_membre(int *nb_membre)
- Pret **creer_tab_pret(int *nb_pret)

```
//.....Initialisation.....
L11: Librairie bibliotheque;
L14: int choix_menu;
...
L44: int continuer = TRUE;
```

Initialisation classique des variables que l'on va retrouver dans la fonction main.

- bibliothèque : Structure qui correspond au cœur du programme, c'est elle qui met en relation tous les livres, membres,...

Architecture de la fonction main

- choix_menu : le choix que l'utilisateur aura demandé dans le menu
- Continuer : variable à TRUE tant que l'utilisateur ne demande pas à quitter le programme

La structure Librairie quant à elle est définie de la façon suivante :

```
typedef struct{
    int donnee[2];  // donnee[0] = nombre de membre depuis l'ouverture de la librairie
                    // donnee[1] = nombre de pret depuis l'ouverture de la librairie
    Donnee_livre donne_livre;
    Liste_admin gestion_admin;
    Liste_membre gestion_membre;
    Liste_livre gestion_livre;
    Liste_pret gestion_pret;
} Librairie;
```

```
typedef struct{
    Membre **liste_membre;
    int nb_membre;
} Liste_membre;
```

```
typedef struct{
    Admin **liste_admin;
    int nb_admin;
} Liste_admin;
```

```
...
```

Pour chaque objets étant des listes, ils sont ajoutés à la variable bibliothèque en 2 étapes :

- calcul_nb_(admin|membre|livre|pret), fonctions définissant le nombre d'admin, membre,... qu'il y aura dans la liste
- creer_tab_(admin|membre|livre|pret), fonctions retournant la liste d'admin membre,...

Une fois la bibliothèque initialisée comme il se doit, l'utilisateur tombe dans une boucle qui, tant que la variable ***continuer*** est a TRUE, va lui demander d'entrer une action, action qui sera ensuite traité dans une condition et qui va appeler la fonction correspondant à cette action.

Quand la condition de la boucle n'est plus remplie, alors on en sort et on quitte le programme une fois toutes les désolations de mémoires faites afin d'éviter toute fuite de mémoire.

CONCLUSION

Dès l'annonce du sujet, nous avons su que ce projet n'allait pas être simple. En effet, bien que l'idée au centre du projet soit simple à comprendre, la mettre en place ne s'annonçai pas si simple. Nous avons donc pris ce devoir comme un défis, le défis de le réussir afin de nous donner la motivation nécessaire pour y parvenir.

Ce projet nous a appris de nombreuses choses comme comment gérer la mise en relation entre différentes données, comment traiter ces dites données de sorte à les garder en mémoire même le programme éteint, et nous a aussi permis de commencer à comprendre le fonctionnement des structures de donnés.

Ces différentes questions et solutions ont permis d'approfondir nos connaissances sur le langage C et le projet en lui-même nous a permis de nous rendre compte qu'une application simple vue de l'extérieur peut se révéler beaucoup plus complexe que l'on pense et que biens que des langages comme Python permettent de faire ce type d'application relativement simplement, l'optimisation impliquant de passer par du C est quant à elle moins simple.

Nous avions pensé à ajouter les fonctionnalités suivantes à l'application :

- ajout de la possibilité d'ajouter des commentaires sur les livres pour mieux conseiller le membre (modification livre).

- ajouter des identifiants et mot de passe pour chaque membre c'est à dire créer de menu un de consultation pour les membres et un avec toutes les fonctionnalités pour les administrateurs.

- Ajout de pénalité d'un membre (au bout de n pénalité, le membre ne peux plus emprunter pendant x semaines).

Cependant, la mise en place de cela n'a pas pu être possible pour cause de temps …