Frydrychowski Valentin
Allanos Mikael

Rapport Projet s4 : ShareThings

Byblio-Tech

I)Introduction

Les fonctions du project on été découpée et placée dans 3 librairies différentes : Json.h, user.h, search.h. La librairie Json.h gère les fichiers de sauvegarde .json. La librairie user.h permet la gestion des utilisateurs et des actions qu'ils peuvent entreprendre. Enfin search.h permet la mise en place d'algorithmes de recherche d'objets suivant différents critères. L'ensemble de ces fonction est utilisé dans un fichier main.c qui sert de menu. Il y a un json par objet et par utilisateur.

Pour représenter les possessions et les emprunts dans la mémoire nous utilisons des tableau de tableau : char** ils sont organisé de la manière suivante

char** pos

char* pos[0]	char* pos[1]	char* pos[2]	char* pos[3]
'3'	'i'	'i'	'i'
'\0'	'd'	'd'	'd'
	'1'	'2'	'3'
	'\0'	'\0'	'\0'

On a donc la première colonne de notre tableau de tableau de char qui indique le nombre de colonne moins 1 donc le nombre d'emprunt ou de possession. La taille des colonne est prédéterminé avec un `#define IDSIZE 64` on a donc des "tableau d'id". Il sont donc alloué de cette manière :

```
char** nv_brw=(char**)malloc(sizeof(char*)*(size+1));
*nv_brw = (char*)malloc(sizeof(char)*(IDSIZE));//pour chacune de colonne(on utilise un for)
```

size étant le nombre de possession ou d'emprunt

Il y a aussi 4 liste qui ont été mise en place "u.json" la liste des utilisateur, "m.json" la liste des mail des utilisateurs, "blacklist.json" la liste des mail des utilisateur ayant été ban et obj.json la liste des objets existants.

II) Json.h

La librairie traite trois type de Json : les utilisateur, les objets et des listes

```
{
    "title" : "Kaamelott l'integral",
    "author" : "Astier Alexandre",
    "date" : 2005,
    "pagenb" : 6969,
    "borrower" : "007",
    "owner" : "j",
    "type" : "comedy"

}
```

ex d'un json d'un objet

```
{
  "forename": "john",
  "name": "Doe",
  "mail": "johngoe@gmail.com",
  "borrowlist": [],
  "possession" : ["1", "444444444"],
  "grade" : 5,
  "pwd" : "azertyuiop"
}
```

ex d'un json d'un utilisateur

```
{
    "mail" : ["clement.truillet@univ-tlse3.fr", "zuck@fb.com"]
}
```

ex d'une liste (banlist)

Les .json sont rassemblés dans le répertoire data de la manière suivante :

```
+- data
|  +- object
|  |  +- 1.json
|  |  +- 444444444.josn
|  |  +- obj.json
|  +- user
|  |  +- blacklist.json
|  |  +- j.json
|  |  +- m.json
|  |  +- u.json
```

- **jsmn.h**

Afin de gérer les différents fichiers json, nous nous sommes aider de la librairie jsmn.h https://github.com/zserge/jsmn . cette libraire vas nous permettre de “parse” le fichier, c’est à dire le découper en plusieurs morceaux afin d’avoir des accesseurs rapide aux différents éléments présents dans le json, que ce soit des chaînes de caractères, tableaux ou encore des entiers. Nous avons choisis celle-ci en particulier car elle est très simple et légère à mettre en oeuvre. En effet elle ne se constitue que du seul .h jsmn.h . Le gros inconvénient de ce parser c’est, et nous nous en somme rendu compte sur le tard, qu’elle exige qu’on lui passe à l’avance le nombre maximum de token qu’elle utilisera, ce qui peut poser des problème dans notre cas et qui dépend de nombre de livres en circulation dans notre bibliothèque. Nous ne nous en somme pas occupé mais on peut imaginer qu’on augmente le nombre maximum de tokens utilisés à chaque ajout de livre.

- **Les fonctions**

- les getters et setters

  Les getters et setters de cette librairie vont tous ouvrir et modifier les json qu’ils gerent grâce à des getteurs et setters gérériques qui prennent en entré le nom du champs et le chemin du json qu’il doivent exploiter

- `int add_us(User user);`

  Permet de creer un nouveau json utilisateur a partir d’une structure utilisateur

- `void add_livre(char*ID, char* title, int pagenb, char* author, int date, char* owner, char* type);`

  Permet de creer un nouveau json objet à partir des informations passé en paramètre

- `int suppr_json(char*path);`

  supprime un fichier dont le chemin est spécifié en paramètre.

- les add_list et suppr list

  tous les add et suppr reposent sur les deux fonctions
  `int supr_List(char* path, char* arg);` et
  `int add_List(char* path, char* arg);`

Elles permettent de modifier des tableaux sous un format json et récupérant les les json sous forme de chaîne de caractères qu’elles analyses et modifie grâce aux fonction de manipulation de string de string.h, pour finalement réécrire la bonne liste dan le fichier modifié.

- `int jsoneq(const char *json, jsmntok_t *tok, const char *s);`
  C'est une fonction essentielle au bon fonctionnement des getters et setters car c'est elle qui vas permettre de trouver le token désignant le champ qui nous intéresse (pour plus d'informations sur `jsmntok_t`, voir la doc de jsmn)

- `int findSize(FILE *fp);`
  Une fonction qui renvoie la taille en nombre de caractères, utile pour allouer la bonne taille à une string qui doit recevoir ce fichier
- `char* jsontochar(char * file_path);`
  Cette fonction permet de récupérer un fichier dont on spécifie le chemin en une chaîne de caractère

- `void chartojson(char * file_path, char * json_text);`
  Celle-ci en revanche permet de faire l'inverse, transformer une chaine de caractère en un fichier.

III) user.h

```
typedef struct user_s {

    char id[IDSIZE]; //ID of a user

    //info user
    char forename[NAMESIZE];
    char name[NAMESIZE];
    char mail[MAILSIZE];
    char** brw; //list of current borrowings
    int grade; //define the amount of power of the user
    char cryptedPw[PWSIZE]; //crypted password
    char** possession;  //list of all possesed books
}user;
```

La librairie user.h repose sur la structure “struc user_s”, dans l’utilisation de cette librairie on passe par un pointeur User pour accéder à la structure. La librairie possède des getters et des setters pour chacun des arguments de la struct. Ex :

Getters et setters :

```
char* uget_id(User util)          void uset_id(char* id, User util)//ne pas free
```

Seul les getters de brw et possession doivent être free car ils retournent des copies

L’objectif de cette librairie et de gérer les utilisateurs et leurs actions. Les utilisateurs peuvent créer un compte, se connecter, déposer et emprunter des objets…

Fonction d’identification :

```
int crea_user(User* util, char* id, char* forename, char* name, char* mail, int grade, char* Pwd)
```

Cette fonction sert à la création d’un utilisateur, elle alloue le pointeur *util et remplie la structure. Ensuite elle appelle add_us(*util) pour créer un json correspondant au nouvel utilisateur et met à jour les liste d’utilisateur et de mail. Code erreur 0 tout va bien, 1 ce mail est blacklisté, 2 une autre utilisateur utilise déjà ce mail, 3 un autre utilisateur utilise déjà cet id.

`int login(User* util,char* id, char* pwd)`

Cette fonction alloue le pointeur *util et remplie la structure avec les données contenue dans le json de l'utilisateur en vérifiant que le mot de passe un fois crypté correspond à celui dans le json. Code erreur 0 tout va bien, 1 le user id n'existe pas, 2 le pwd ne correspond pas.

`void logout(User user)`

Cette fonction libère la structure dont le pointeur est passé en paramètre.

Fonction d'emprunt dépôt et retour :

`int borrowing(User util, char* idObject)`

Cette fonction permet à l'utilisateur passé en paramètre d'emprunter un objet passé en paramètre. La fonction crée une copie du brw(char**) de l'utilisateur en y ajoutant le nouvel emprunt. Ensuite la fonction met à jour la structure et les json de l'objet et de l'utilisateur. Code erreur 0 tout va bien, 1 l'objet a déjà un borrower.

`int return_back(char* idObject, User util)`

Cette fonction permet à l'utilisateur passé en paramètre de rendre un objet passé en paramètre. La fonction crée une copie du brw(char**) de l'utilisateur en y enlevant l'objet rendu. Ensuite la fonction met à jour la structure et les json de l'objet et de l'utilisateur. Code erreur 0 tout va bien, 1 l'objet n'a pas été emprunter par cette utilisateur.

`void return_back_all(User util)`

Cette fonction permet à l'utilisateur passé en paramètre de rendre tout les objet emprunté. La fonction créer un bwr vide(uniquement avec le premier champs=0). Ensuite les json de tous le objets sont mis à jour puis la structure et le json de l'utilisateur sont mis à jour.

`void add_possession(User user,char* name, int pagenb, char* author, int date, char* kind)`

Cette fonction permet à l'utilisateur passé en paramètre de déposer un objet. D'abord la fonction crée l'id de l'objet à partir de la date et du temps en seconde. Ensuite elle crée une copie de possession de user et y ajoute l'id créée. Enfin elle crée un json pour l'objet avec `add_livre(idObject, name, pagenb, author, date, user->id,kind)` puis met à jour la liste des objets, la structure de user et le json de user.

`int suppr_possession(char* idObject, User user)`

Cette fonction permet à l'utilisateur passé en paramètre de récupérer une possession passée en paramètre. La fonction crée une copie de possession de user sans idObject puis met à jour le json et la structure de l'utilisateur. Enfin le json de l'objet est supprimé et la liste de object est mise à jour. Code erreur 0 tout va bien, 1 l'objet est emprunté par quelqu'un, 2 l'utilisateur ne possède pas cet objet.

`void suppr_all_possession(User user)`

Cette fonction permet à l'utilisateur passé en paramètre de récupérer toute ces possession(est utilisé lors des procédures de ban uniquement). D'abord la fonction fait rendre les possessions empruntés par les autres utilisateurs (donc en mettant à jour leur borrowlist) puis supprime les json de tous les possession. Enfin elle crée un tableau de possession vide et update la structure et le json de l'utilisateur.

Fonction de gestion des utilisateur

`int suppr_us(User user)`

Cette fonction permet de supprimer le compte d'un utilisateur si ces possessions ne sont pas emprunté. La fonction supprime toutes les possessions de l'utilisateur, rend tout ces emprunts, met à jour les listes de mail et de user et puis supprime le json de l'utilisateur. Code erreur 0 tout va bien, 1 une des possession a été empruntée par un utilisateur.

`int ban(char* id,User user)`

Cette fonction à un utilisateur user de ban un autre utilisateur id si il en a les droits. La fonction supprime toutes les possessions de l'utilisateur en mettant à jour les json des utilisateur ayant emprunté une possession de id. Ensuite rend tout ces emprunts, met à jour les listes de mail, de user et blacklist et puis supprime le json de l'utilisateur. Code erreur 0 tout va bien, 1 l'utilisateur id n'existe pas, 2 le grade de id est supérieur ou égale à celui de user.

`int new_pwd(User user, char* pwd, char* nv_pwd)`

Cette fonction permet à un utilisateur de changer son mot de passe. pwd le mot de passe actuel est testé puis remplacé par nv_pwd crypté. Code erreur 0 tout va bien, 1 pwd est incorrecte

`int new_username(User user,char* new_username)`

Cette fonction permet à une utilisateur de charger son username(id). La fonction crée une copie du json en changeant le nom et y copie les information du json initial. Ensuite la liste de user est mise à jour et le json initiale supprimé. Code erreur 0 tout va bien, 1 new_username est déjà pris par un utilisateur.

`int change_grade(User user,char* id,int newgrade)`

Cette fonction permet à un utilisateur user de changer le grade d'un autre utilisateur id si il en a les droits. Code erreur 0 tout va bien, 1 le grade de user est inférieur ou égale à celui de id, 2 si newgrade est supérieur au grade de au grade de user.

Fonction utilitaire:

`int exist_user(char* id)`

Cette fonction return 0 si l'utilisateur id possède un json sinon return 0.

`int possession_borrow(User user)`

Cette fonction return le nombre de possession de user qui sont empruntés.

`int exist_in_list(char* substring,char* listname)`
Cette fonction return 0 si il y a au moins une occurrence de subsstring dans la liste listename sinon return 0. La liste doit se trouver dans user.

`User charge_user(char* id)//TO free`
Cette fonction return un pointeur vers une structure struct user_s ayant été remplie avec les données du json de id.

`void free_user(User user)`
Cette fonction libère la structure struct user_s pointée par le pointeur passé en paramètre.

`int exist_in_table(char* id,char** table)`
Cette fonction return le nombre d'occurrence de id dans table

`void encrypt(char* pwd,char* crypwd)`
Cette fonction crypte pwd dans crypwd. La clé pour le cryptage est calculé avec la somme des codes ascii des caractères(-32). Ensuite le cryptage fonctionne comme un cryptage de césar en code ascii entre 32 et 127.

`char** duplicate_table(char** tab)//TO free`
Cette fonction copie le tableau de tableau passé en entrée et return le pointeur de la copier

IV) Search.h

- `void search(User us);`

La fonction la plus importante de cette bibliothèque, car c'est autour d'elle que s'articule toute les autres. Elle intègre son propre menu indépendant du menu présent dans main.h, si ce n'est qu'elle a besoin du user qui effectue la recherche afin de pouvoir lui ajouter des emprunts .

- `void sub_searchM(int* pos, int* query, char** index, User us);`

Cette fonction ne sert qu'à éviter la redondance de code dans la fonction précédente. Elle gère la partie d'affichage des résultats de recherche et le menu d'emprunt qui résulte de cette recherche

- `char** search_title(char* name);`
- `char** search_author(char* author);`
- `char** search_date(char* date);`
- `char** search_type(char* type);`

Ces quatres fonctions permettent de rechercher des objets ainsi que de les lister selon une des caractéristiques. Elle sortent un tableau de chaînes de caractères qui contient dans les cases paires les id des objets, et dans les cases impaires les caractéristiques de ces objets, ceci afin de ne pas séparer la caractéristique de l'objet lors de son tri. Nous somme conscient de la forte redondance de ces quatres fonctions mais c'était un moyen simple de les écrires car elles ont toutes les quatres de petites différences notable en plusieurs endroits de leur codes. A noter qu'il est possible de rechercher aussi un objet par son ID précise, mais cette recherche est directement implanté dans la fonction search

- `int cstring_cmp(const void *a, const void *b);`

Cette fonction est une réécriture de strcmp présente dans string.h afin de pouvoir l'utiliser dans la fonction qsort de stdlib.h. Celle ci est censé pouvoir nous permettre de trier les tableaux de string par ordre alphabétique. Ici cstring_cmp joue le rôle de comparateur de chaînes de caractère.

V) Guide d'installation de Byblio-Tech

- 1 : rendez-vous sur https://github.com/vfrydrychowski/Byblio-Tech
- 2 : récupérez tous les fichier
- 3 : A la racine du projet, faite un make
- 4 : lancer Byblio-Tech.exe

VI) Dificultées

- La distance

  Même si la programmation n'est pas une discipline qui requiert forcément le contact, nous pensons que celle-ci a nui à notre productivité et a mené à des confusion qui nous ont fait perdre pas mal d'heures car nous n'étions pas au claire sur des fonctions basiques

- Dépendance des librairies

  Les dépendances des librairies rend la recherche et la correction des erreur beaucoup plus complexes car avec la répartition des fonctions entre nous un problème lors de l'implémentation d'un fonction peut venir d'une fonction que l'autre a codé. Il faut donc le consulter et attendre qu'il soit disponible pour trouver l'erreur.

- Prévoir le temps de travail :

  la gestion du temps est globalement un échec, nous avons dû rattraper en très peu de temps le retard accumulé et cela a mené à l'abandon de certaines fonctionnalitées de notre programme