TREMBLAY Jérémy
WISSOCQ Maxime
COUDOUR Adrien

Compte Rendu du Projet de Structures P2

Vous trouverez dans ce document nos structures, nos fonctions, leurs actions, les fichiers, la répartition des tâches concernant ce projet.
Bonne lecture.

Sommaire

I – Les fichiers

- 1.1 Le fichier des jeux
- 1.2 Le fichier des adhérents
- 1.3 Le fichier des emprunts
- 1.4 Le fichier des réservations
- 1.5 Les fichiers des programmes
  - 1.5.1 donnees.c
  - 1.5.2 menu.c
  - 1.5.3 recherchesEtTris
  - 1.5.4 ludotheque.c
  - 1.5.5 ludotheque.h
  - 1.5.6 ludotheque.c

II – Les structures et données

- 2.1 Les jeux
- 2.2 Les adhérents
- 2.3 Les emprunts
- 2.4 Les réservations

III – La liste des fonctionnalités

- 3.1 Le chargement des données
- 3.2 L'affichage des jeux disponibles
- 3.3 L'affichage des emprunts en cours
- 3.4 L'affichage des réservations
- 3.5 La création d'un emprunt/réservation et/ou la création d'un adhérent
- 3.6 Le retour d'un jeu
- 3.7 La suppression d'une réservation
- 3.8 La sauvegarde des données

IV – La répartition du travail

Bonne lecture.

I – Les fichiers

Durant ce projet, nous avons longuement réfléchis à la manière dont nous allons charger, enregistrer nos données, et les traiter. Nous avons décidé de réaliser des fichiers binaires pour les données des Jeux, des emprunts et des adhérents, car les fichiers binaires sont beaucoup plus rapides à manipuler que les fichiers réguliers (traitements très lourd, très coûteux en temps de calculs).

Pour les réservations nous avons décidés de charger et d'enregistrer nos données depuis un fichier régulier classique.

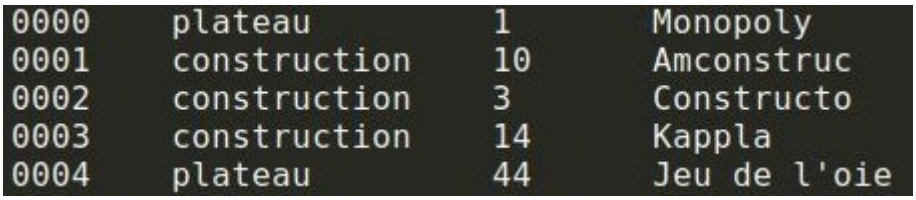
Quoi qu'il advienne, même si nous voulions charger / sauvegarder nos données depuis un fichier binaire, il faudrait tout de même réaliser un fichier régulier où seront lues les données une fois (une seule fois, lors de l'insertion de premières données, comme par exemple les jeux), pour être ensuite enregistrées dans un fichier binaire.

Nous allions donc avoir 3 fichiers réguliers (pour les jeux, les adhérents, et les emprunts) dont les valeurs à l'intérieur ne changeront pas (les données de départ), et un fichier de réservation régulier où seront chargées / enregistrées les données des réservations à chaque lancement / fin de programme.

1.1 Le fichier des jeux

Ce fichier est essentiel dans la ludothèque : sans jeux, impossible de faire des emprunts / réservations. Nous l'avons donc initialisé avec quelques valeurs au départ, puis nous l'avons chargé une première fois, nous ne l'avons pas modifié par la suite.

Voici son contenu :



Une partie du contenu du fichier jeu.don

Comme on peut le voir sur l'image, de gauche à droite, les données sont les suivantes :

- L'ID du jeu (entier),
- Le type du jeu (plateau, construction, tuile, logique ou carte), chaîne de caractères,
- Le nombre d'exemplaires du jeu ou stock (entier),
- Le nom du jeu (chaîne de caractères).

On remarque que toutes les données d'un jeu se trouvent sur une seule et même ligne. On peut faire quelques précisions concernant son contenu :

- L'ID du jeu doit être **UNIQUE** (=pas de doublons). Il doit être supérieur ou égal à 0,
- Le type du jeu doit être compris dans les valeurs énoncées au-dessus,
- Le stock doit être supérieur ou égal à 0,

- Le nom du jeu peut-être composé,
- Deux jeux ne doivent pas porter le même nom,
- **Les valeurs peuvent se trouver dans n'importe quel ordre**, cela n'a aucune importance.

1.2 Le fichier des adhérents

Ce fichier contient les données de chaque personne de la ludothèque (informations personnelles). Voici le contenu de ce fichier :

```
0000        01/02/2020      Mme     Michelle        DEBOUT
0001        04/10/2020      Mr      Martin          DE CHANTAIL
0002        10/12/2019      Mme     Francoise       DE FOURVOIS
0003        15/07/2020      Mr      Roger           VERNIN
0004        02/03/2017      Mr      Marcel          LORGEOUX
```

Une partie du contenu du fichier adherent.don

Les données de gauche à droite sont les suivantes :

- L'ID de l'adhérent (entier),
- La date de son inscription (elle est composée d'un jour, d'un mois, et d'une année, entiers),
- La civilité de la personne (Mr ou Mme, chaîne de caractères),
- Le prénom de la personne (chaîne de caractères),
- Le nom de la personne (chaîne de caractères).

On remarque que toutes les données qui concernent un adhérent se trouvent sur une seule et même ligne. On peut faire quelques précisions concernant le contenu de ce fichier :

- L'ID d'adhérent doit être **UNIQUE** (=pas de doublons). Il doit être supérieur ou égal à 0.
- La date d'inscription doit être inférieure ou égale à la date du jour,
- La civilité doit être soit « Mme », soit « Mr »,
- Le prénom n'est pas composé.
- Le nom peut être composé.
- On peut avoir des noms / prénoms / dates d'inscriptions similaires dans la base de données.
- **Il doit être trié en fonction de l'ID des adhérents (ordre croissant).**

1.3 Le fichier des emprunts

Celui-ci contient les données relatives aux emprunts : qui a emprunté quoi, quand… Voici à quoi il ressemble :

```
0001        0000        0001        01/12/2020
0003        0004        0011        04/12/2020
0004        0001        0004        08/12/2020
0000        0001        0006        09/12/2020
0002        0003        0001        09/12/2020
```

Une partie du contenu du fichier emprunt.don

Les données de gauche à droite sont les suivantes :

- L'ID de l'emprunt (entier),
- L'ID de l'adhérent qui a réalisé l'emprunt (entier),
- L'ID du jeu qui a été emprunté (entier),
- La date de l'emprunt (elle est composée d'un jour, d'un mois, et d'une année, entiers).

On remarque que toutes les données qui concernent un emprunt se trouvent sur une seule et même ligne. On peut faire quelques précisions concernant le contenu de ce fichier :

- L'ID d'emprunt doit être **UNIQUE** (= pas de doublons), il peut aussi être **manquant** (pas dans le fichier),
- L'ID de l'adhérent peut se trouver **plusieurs fois** dans les emprunts (un adhérent peut emprunter 3 fois), mais **il doit être référencé dans le fichier des adhérents** (il doit exister).
- L'ID du jeu emprunté peut se trouver **plusieurs fois** dans les emprunts (un jeu peut-être emprunté plusieurs fois en même temps), mais **il doit être référencé dans le fichier des jeux** (il doit exister).
- La date de l'emprunt doit être inférieure ou égale à la date du jour,
- **Les données de ce fichier sont triées par date d'emprunt (croissant)**.

1.4 Le fichier des réservations

Ce fichier contient les données de chaque réservation de la ludothèque : qui a réservé quoi, quand... Ce fichier est chargé et modifié lors de la sauvegarde à chaque lancement / fin de programme. Voici le contenu de ce fichier :

```
0000        0001        0012        03/05/2020
0001        0000        0021        25/11/2020
0003        0000        0012        17/10/2019
0004        0003        0013        12/12/2020
```

Une partie du contenu du fichier réservations.don

Les données de gauche à droite sont les suivantes :

- L'ID de la réservation (entier),
- L'ID de l'adhérent qui a réalisé la réservation (entier),
- L'ID du jeu qui a été réservé (entier),
- La date de la réservation (elle est composée d'un jour, d'un mois, et d'une année, entiers).

On remarque que toutes les données qui concernent une réservation se trouvent sur une seule et même ligne. On peut faire quelques précisions concernant le contenu de ce fichier :

- L'ID de réservation doit être **UNIQUE** (= pas de doublons), il peut aussi être **manquant** (pas dans le fichier),
- L'ID de l'adhérent peut se trouver **plusieurs fois** dans les réservations (un adhérent peut réserver plusieurs fois), mais **il doit être référencé dans le fichier des adhérents** (il doit exister).

- L'ID du jeu emprunté peut se trouver **plusieurs fois** dans les réservations (un jeu peut-être réservé plusieurs fois en même temps), mais **il doit être référencé dans le fichier des jeux** (il doit exister).
- La date de réservation doit être inférieure ou égale à la date du jour,
- **Les données de ce fichier ne sont pas triées.** Elles peuvent se trouver dans n'importe quel ordre (même si sur l'image on voit les réservations triées par ID, ce n'est pas obligatoire).

1.5 Les fichiers des programmes

Nous allons ici nous attarder sur les différents fichiers qui composent le programme. Nous avons découpé notre programme en plusieurs fichiers afin de le rendre plus lisible et de s'y retrouver plus facilement. Plutôt que de faire un découpage classique « en fonction des structures », nous l'avons découpé par types de fonctions.

1.5.1 donnees.c

Ce fichier contient les fonctions de chargement, de lecture d'une ligne, et de sauvegarde (depuis les fichiers binaires et réguliers). On retrouve aussi les fonctions d'insertion dans le cas de la liste chaînée, de, de création de liste vide, de libération de la mémoire (pour la liste chaînée). Il y a aussi un affichage de débogage pour chaque type de données (affiche le contenu des tableaux / liste).

Il couvre ainsi les fonctionnalités 1 et 8.

1.5.2 menu.c

Ce fichier contient les fonctions du menu, du sous menu d'affichage, et du sous-sous menu d'affichage des jeux disponibles. Par ailleurs, il contient les fonctions d'affichage de ces menus. On y retrouve aussi la fonction globale qui charge les données, appelle le menu, enregistre les données et libère la mémoire.

1.5.3 rechercheEtTris

Comme son nom l'indique, ce fichier contient les fonctions de tri (tri jeux par nom, emprunt par ID, et emprunt par date d'emprunt), ainsi que différentes fonctions de recherche.

1.5.4 ludotheque.c

Ce fichier contient les fonctions principales de notre programmes, celles qui sont appelées depuis le menu. Il contient aussi les autres fonctions d'ajout d'emprunt, d'adhérent, de suppression, de comparaison de dates...

1.5.5 ludotheque.h

Ce fichier contient les déclarations de nos fonctions, elles sont triées par fonctionnalités. On y retrouve aussi nos structures (présentées dans la partie suivante).

1.5.6 ludotheque.c

Dans ce fichier, on peut retrouver la fonction principale *main*, ainsi que les fonctions de tests.

II – Les structures et données

2.1 Les jeux

Nous avons donc décidé de créer une structure correspondant aux données qui se trouvent dans le fichier des jeux comme dit plus haut, à savoir : un ID, un nom de jeu, un type de jeu, un nombre d'exemplaire. Voici la structure d'un jeu correspondante :

```
typedef struct
{
    int idJeu;
    char nomJeu[30];
    char typeJeu[13];
    int nbExemplaire;
} Jeu;
```

A gauche, la structure d'une variable de type Jeu.

Le type du jeu fait au maximum 13 caractères (avec l'« \0 »).

Nous avons donc décidé de créer non pas un tableau de Jeux… Mais un tableau de pointeurs, car il permet d'éviter de déplacer des données trop volumineuses (dans le cas où on aura des centaines de jeux ce sera indispensable!), et l'allocation de données se fera ainsi de manière continue, avec des allocations dynamiques.

Nous avons donc déclaré notre tableau de pointeurs de cette manière :

```
Jeu *tJeux[TAILLEMAX];
```

Déclaration d'un tableau de pointeurs sur une structure Jeu de taille TAILLEMAX.

TAILLEMAX étant une constante déclarée qui vaut 10000.

2.2 Les adhérents

Concernant les adhérents, nous voulions un moyen simple et rapide d'insérer un adhérent. Dans le cadre de ce projet nous ne supprimons jamais d'adhérents de nos données et les adhérents sont triés par ID. Ainsi, la création d'un adhérent va s'en trouver simplifiée (la recherche d'un ID disponible). Nous auront juste à dire que l'ID d'un nouvel adhérent (qui s'inscrit) sera égal à la taille logique (les ID commençant à 0).

Nous avons donc également opté pour un tableau de pointeurs.

Il nous a fallu créer une structure Date pour toutes les dates de ce projet (dont notamment la date d'inscription) :

```
typedef struct
{
    int jour;
    int mois;
    int annee;
} Date;
```

Une structure Date composée d'un jour, d'un mois, et d'une année !

Nous avons songé a utiliser *enum* pour énumérer des valeurs de mois que nous aurions mis avec des chaînes de caractères dans les fichiers, mais nous avons trouvé ce choix plus simple et plus en accord avec les cours de structures de cette période.

Nous avons ensuite créé notre structure d'adhérent :

```
typedef struct
{
    int idAdherent;
    char civilite[4];
    char nom[30];
    char prenom[30];
    Date dateInscription;
} Adherent;
```

La structure d'un Adhérent : comprend un ID, une civilité (Mr /Mme), un nom, un prénom et une date d'inscription.

2.3 Les emprunts

Pour les emprunts, nous avons appliqué le même raisonnement. Nous avons utilisé un tableaux de pointeurs (plus simple pour la manipulation des données), et créé une structure d'emprunt:

```
typedef struct
{
    int idEmprunt;
    int idAdherent;
    int idJeu;
    Date dateEmprunt;
} Emprunt;
```

Une structure d'emprunt comportant un ID de jeu, d'adhérent et d'emprunt, ainsi qu'une date d'emprunt.

2.4 Les réservations

Pour les réservations nous avons réfléchis différemment. Nous nous sommes dis que ce serait intéressant de manipuler la récursivité et les listes, et donc il nous semblait pertinent de faire une liste chaînée pour enregistrer les données des réservations.

Nous avons donc créé une structure réservation :

```
typedef struct
{
    int idResa;
    int idAdherent;
    int idJeux;
    Date dateReservation;
} Reservation;
```

Une réservation est une variable comportant un ID de jeu, d'adhérent, de réservation (unique!) et une date de réservation.

Ensuite nous avons créé notre liste de réservation. Cette liste se compose d'une réservation, et d'un pointeur sur l'élément du maillon suivant de la liste. Ensuite on définie la liste comme étant un pointeur de Maillon :

```
typedef struct liste
{
    Reservation r;
    struct liste *suivant;
} Maillon, *ListeReservation;
```

Création d'une liste chaînée.

C'est ainsi que nous avons mis en place la manière dont seront sauvegardées nos données.

Nous avons par la suite commencé à mettre en place les fonctionnalités demandées.

III – La liste des fonctionnalités

Toutes les fonctionnalités proposées fonctionnent normalement. Nous allons ici expliquer leur fonctionnement plus en détail.

3.1 Le chargement des données

Au début nous devions créer une fonction de chargement pour les 4 types de données. Ainsi, il fallait charger les données des jeux. Pour ce faire nous avons réalisé une fonction qui ouvre le fichier créé précédemment, il appelle une fonction qui lit une ligne et qui enregistre les données dans une variable de type jeu. Ensuite on vient allouer dynamiquement une place dans notre tableau de pointeurs et ajouter le jeu lu. On fait ça jusqu'à la fin du fichier.

Le principe est le même pour toutes les données que l'on enregistre dans nos tableaux de pointeurs (adhérents, jeux et emprunts). Donc le programme des deux autres chargements reste très similaire (on a toujours une fonction qui lit une ligne et retourne l'élément lu, et la fonction plus générale qui charge le fichier et rempli le tableau).

Nous voulions travailler avec les fichiers binaires pour ces 3 tableaux de pointeurs. Seulement pour ce faire, il fallait déjà enregistrer les données dans un fichier binaire, puis les charger depuis ce fichier. Il nous fallait donc faire une fonction de sauvegarde dans un fichier binaire (voir la partie 3.8 La sauvegarde des données), puis une autre fonction de chargement depuis un fichier binaire.

Nous avons donc réalisé une fonction de chargement depuis un fichier binaire pour nos trois tableaux de pointeurs. La fonction est similaire la même que le chargement précédent, la seule différence c'est que nous n'avons pas fait de fonction qui lit une ligne, on utilise directement *fread* et on place le jeu lu dan le tableau de pointeurs :

On charge le fichier binaire cette fois-ci, qui porte l'extension « .bin »:

```
fe = fopen("jeu.bin", "rb");
```

On lit avec fread *au lieu de* fscanf.

```
fread(tJeux[i], sizeof(Jeu), 1, fe);
```

Nous avons réalisé cette fonction de chargement depuis un fichier binaire pour nos trois tableaux de pointeurs.

Pour notre liste chaînée de réservations, la méthode était différente. Nous devions créer une fonction qui crée une liste vide (retourner NULL), une fonction de chargement. Le problème c'est que c'est une liste que nous avions, et il fallait donc utiliser une fonction récursive lors de l'insertion (ajout d'une réservation).

Nous avons réalisé deux fonctions : une fonction qui insère en tête de liste (on crée un maillon, on lui donne une réservation, et on le fait pointer sur le reste de la liste, et on le retourne), et une fonction qui va s'occuper d'insérer la réservation au bon endroit (on parcourt de manière récursive notre liste et si on trouve une place disponible on appelle l'autre fonction d'insertion, on retourne la liste modifiée).

Nous avons décidé de trier notre liste chaînée par ID de réservation, nous nous sommes dis que ce serait plus simple lors de l'insertion d'un emprunt (lorsqu'on recherche un ID, on recherche un « trou » dans la liste chaînée, un ID qui n'existe pas déjà) et lors d'une suppression (on parcourt jusqu'à trouver l'ID correspondant).

Voici donc nos fonctions d'insertions :

```
ListeReservation InsertionEnTete(ListeReservation lr, Reservation reserv)
{
    Maillon *mail;
    mail = (Maillon *) malloc(sizeof(Maillon));
    if (mail == NULL)
    {
        printf("Problème d'allocation dynamique avec malloc dans les donn
        return NULL;
    }
    mail -> r = reserv;
    mail -> suivant = lr;
    return mail;
}
ListeReservation ajouterMaillon(ListeReservation lr, Reservation reserv)
{
    if (lr == NULL)
        return InsertionEnTete(lr, reserv);
    if (lr->r.idResa > reserv.idResa)
        return InsertionEnTete(lr, reserv);
    lr->suivant = ajouterMaillon(lr->suivant, reserv);
    return lr;
}
```

En haut : l'insertion en tête qui ajoute juste une réservation fournie en paramètre, et qui retourne une nouvelle liste.
En bas : l'insertion qui va permettre d'enregistrer les données triées par ID de réservation croissante dans la liste chaînée.

Nous avons par la suite réalisé une fonction qui lit une ligne du fichier des réservations, qui retourne une réservation contenant ces informations. Enfin, nous avons fait notre fonction de

chargement, elle est basique. On ouvre le fichier et on appelle la fonction de lecture afin de récupérer la réservation lue qu'on envoie à la fonction d'insertion qui va l'ajouter à la liste, dans l'ordre des ID:

```
ListeReservation chargementReservation(ListeReservation lr)
{
    Reservation reserv;
    FILE *fe;

    fe = fopen("reservations.don", "r");
    if (fe == NULL)
    {
        printf("Problème lors de l'ouverture du fichier res
        return NULL;
    }

    reserv = lireReservation(fe);

    while(feof(fe) == 0)
    {
        lr = ajouterMaillon(lr, reserv);

        reserv = lireReservation(fe);
    }

    fclose(fe);
    return lr;
}
```

La fonction de chargement des réservations.

Par la suite nous sommes passés aux affichages qui étaient demandés.

3.2 L'affichage des jeux disponibles

Pour commencer à mettre en œuvre les choix que pouvait faire l'utilisateur, on a fait les menus. Nous avons décidé de faire un menu global, et un sous menu accessible à ce menu où l'on pourrait faire les affichages des fonctionnalités 2, 3, 4. Il aurait aussi un sous menu pour les jeux, afin d'afficher les jeux disponibles par type de jeux, ainsi que tous les jeux.

Nous avons réalisé une fonction globale qui fait le chargement des données, appelle le menu, et enregistre les données dans les fichiers binaires (le fichier régulier pour la liste) en fin de traitement, et libère la mémoire.

Pour afficher les jeux disponibles triés par ordre alphabétique de leur nom, nous devions tout d'abord réaliser une fonction qui les trie par ordre alphabétique, et c'est ce que nous avons fait :

```c
int rechercheRMin(Jeu *tJeux[], int i, int n)
{
    int rmin = i, j;
    for (j = i + 1; j < n; j++)
    {
        if (strcmp(tJeux[j]->nomJeu, tJeux[rmin]->nomJeu) < 0)
            rmin = j;
    }
    return rmin;
}
```

Fonction de recherche d'un rang minimum.

```c
void permutation(Jeu *tJeux[], int i, int j)
{
    Jeu permutation;
    permutation = *tJeux[i];
    *tJeux[i] = *tJeux[j];
    *tJeux[j] = permutation;
}
```

Fonction de permutation de deux jeux.

```c
void triJeuxParNom(Jeu *tJeux[], int nbJeux)
{
    int k, rmin;
    for (k = 0; k < nbJeux - 1; k++)
    {
        rmin = rechercheRMin(tJeux, k, nbJeux);
        permutation(tJeux, k, rmin);
    }
}
```

Fonction principale de tri.

Nous avons réalisé un tri par sélection / échange. Une fonction recherche le rang minimum du tableau et retourne sa position, la fonction principale parcourt le tableau, récupère le rang minimum, et appelle la fonction de permutation.

Maintenant, nous avons décidé d'appeler cette fonction de tri dans la fonction globale, juste après le chargement des jeux. Comme les jeux ne sont pas triés, autant les trier dès le chargement (c'est pas grave s'ils sont enregistrés triés par nom, ça ne va rien changer).

Le problème était que nous devions aussi les trier par type de jeu auparavant. Nous ne voulions pas recréer des tableaux en insérant les jeux d'un type à l'intérieur, puis les trier, les afficher… Donc nous avons fait une fonction qui affiche uniquement les jeux d'un type (fourni en paramètre). Voici ce qu'elle donne :

```c
int affichageJeuxDunType(Jeu *tJeux[], int nbJeux, char *typeJeu)
{
    int i, compteur = 0;

    for (i = 0; i < nbJeux; i++)
    {
        if (strcmp(tJeux[i]->typeJeu, typeJeu) == 0 && tJeux[i]->nbExemplaire != 0)
        {
            printf("%-10s\t\t%-30s\t\t%d\n", tJeux[i]->typeJeu,
                tJeux[i]->nomJeu, tJeux[i]->nbExemplaire);
            compteur++;
        }
    }
    return compteur;
}
```

Affichage des jeux du type fourni en paramètre.

Cette fonction parcours juste le tableaux de pointeurs des jeux et si un jeu est du type donné et qu'il est empruntable (que son stock est supérieur à 0), on l'affiche. Elle retourne le nombre d'affichages.

Comme les jeux sont déjà triés par nom, cela les affiche bien triés par type et par nom.

Nous n'affichons pas les ID, nous trouvions cela inutile dans notre cas. Les utilisateurs n'auront qu'à entrer le nom du jeu qu'ils voudront emprunter, ce sera plus simple que de retenir une valeur.

Par la suite nous avons fait la fonction qui affiche tous les types de jeux. Si la fonction précédente n'a rien affichée alors on retourne un petit message disant qu'aucun jeu existe pour le moment :

```c
void affichageJeuxDisponibles(Jeu *tJeux[], int nbJeux)
{
    int compteur = 0;
    printf("\nTYPE\t\t\tNOM\t\t\t\tSTOCK\n");

    compteur = affichageJeuxDunType(tJeux, nbJeux, "carte");
    compteur = affichageJeuxDunType(tJeux, nbJeux, "construction") + compteur;
    compteur = affichageJeuxDunType(tJeux, nbJeux, "logique") + compteur;
    compteur = affichageJeuxDunType(tJeux, nbJeux, "plateau") + compteur;
    compteur = affichageJeuxDunType(tJeux, nbJeux, "tuile") + compteur;

    if (compteur == 0)
        printf("Il ne semble pas y avoir de jeux disponibles pour le moment.\n");

    printf("\n");
}
```

Affichage de tous les jeux de tous les types.

3.3 L’affichage des emprunts en cours

Pour l’affichage des emprunts, nous devions – avant de le commencer – faire des fonctions de recherche. En effet, nous rappelons que dans le fichier des emprunts (et aussi des réservations au passage), nous avons des ID de jeux, des ID d’adhérents, mais pas leurs infos (nom du jeu, de l’adhérent, civilité, etc). Il nous fallait des fonctions qui parcourent donc les autres tableaux afin de nous donner ces informations.

Nos fonctions sont des recherches itératives : on parcourt le tableau, si on trouve que l’id d’un élément du tableau (adhérent, jeu…) correspond avec celui donné en paramètre, il s’agit de la bonne valeur. A ce moment là, on copie toutes les informations concernant cet élément dans une nouvelle structure qu’on aura fournie en pointeur. La fonction retourne -1 si elle n’a pas trouvée la valeur (cela servira à afficher des messages d’erreurs), ou la position à laquelle il se trouvait.

Nous avons fait une fonction qui recherche l’identité d’un adhérent à partir d’un id. Si on trouve l’ID donné dans le tableau, alors on récupère les informations de l’adhérent correspondant dans une structure :

```c
int rechercheIdentiteAdherent(Adherent **tAdh, int nbAdh, int idAdherent, Adherent *adh)
{
    int i;

    for (i = 0; i < nbAdh; i++)
    {
        if (tAdh[i]->idAdherent == idAdherent)
        {
            //On copie le tout dans la structure passée par pointeur.
            strcpy(adh->nom, tAdh[i]->nom);
            strcpy(adh->prenom, tAdh[i]->prenom);
            strcpy(adh->civilite, tAdh[i]->civilite);
            adh->dateInscription.jour = tAdh[i]->dateInscription.jour;
            adh->dateInscription.mois = tAdh[i]->dateInscription.mois;
            adh->dateInscription.annee = tAdh[i]->dateInscription.annee;
            adh->idAdherent = tAdh[i]->idAdherent;
            return i;
        }

        //Comme les adhérents sont triés par ID, cela signifie qu'on l'a dépassé.
        if (tAdh[i]->idAdherent > idAdherent)
            return -1;
    }
    return -1;
}
```

Fonction de recherche des informations d’un adhérent.

Nous aurions pu faire une recherche dichotomique, mais par manque du temps nous avons du faire une recherche classique.

Nous avons fait 2 fonctions de recherche à propos des jeux. On récupère les informations d’un jeu soit en fournissant un ID (le cas des emprunts où on a qu’un ID dans le fichier), soit en fournissant un nom (dans le cas des saisies).

La fonction d’affichage parcours le tableau des emprunts, appelle ces fonctions afin de récupérer des données, puis affiche le tout (jeu emprunté, emprunteur, date…) :

```c
int affichageEmpruntEnCours(Jeu **tJeux, int nbJeux, Adherent **tAdh, int nbAdh, Emprunt **tEm
{
    int i, erreur;
    Jeu jouet;
    Adherent adh;

    //Parcour des emprunts.
    for(i = 0; i < nbEmp; i++)
    {
        erreur = rechercheIdentiteAdherent(tAdh, nbAdh, tEmp[i]->idAdherent, &adh);
        if (erreur == -1)           //Ne devrait JAMAIS arriver, seulement si un adhérent à ét
            return i;

        erreur = rechercheInfosJeu2(tJeux, nbJeux, tEmp[i]->idJeu, &jouet);
        if (erreur == -1)                       //Ne devrait JAMAIS arriver, seulement si un j
            strcpy(jouet.nomJeu, "Jeu inexistant");

        //Affichage.

        printf("\nDate d'emprunt : %02d/%02d/%04d\nNom du jeu : %s\nEmprunteur : %s %s %s\n",
            tEmp[i]->dateEmprunt.annee, jouet.nomJeu, adh.civilite, adh.nom, adh.prenom);
    }
    return i;
}
```

Elle retourne le nombre d'affichages.

La fonction principale appelée depuis le menu ne fait qu'appeler cette fonction d'affichage, et affiche un message comme quoi il n'y a pas d'emprunts si l'autre fonction n'a rien affiché.

3.4 L'affichage des réservations

Pour mettre en œuvre cette fonctionnalité, nous allions avoir besoin de notre liste de réservation, et nous avions déjà programmé nos fonctions de recherche auparavant.

La fonction principale appelée depuis le menu, est celle qui demande les saisies à l'utilisateur. Il doit entrer le nom de jeu duquel il veut voir les réservations.

Comme le nom de jeu peut être composé, on utilise *fgets*. On appelle notre fonction de recherche, et si on ne le trouve pas (jeu entré inconnu), on affiche un message:

```c
erreur = rechercheInfosJeux(tJeux, nbJeux, nomJeu, &jeuChoisi);

if (erreur == -1)
{
    printf("Le jeu entré n'existe pas. Vérifiez l'orthographe et réessayez.\n");
    return;
}
```

Un message d'erreur.

Si le jeu existe, on appelle notre fonction d'affichages de réservations. Elle retourne le nombre d'affichages, donc si ce nombre vaut 0, on affiche un message pour dire qu'il n'en existe pas pour ce jeu.

Mais nous avons du créer cette fonction d'affichage récursive (à cause de la liste chaînée) !

On lui envoie l'ID du jeu que la personne a donnée (lorsqu'elle a entrée son nom), et on parcours la liste. Si on trouve un ID de jeux d'une réservation correspondant au même ID donné, alors bingo ! On a une réservation pour ce jeu ! Alors on appelle la fonction qui va récupérer les informations de l'adhérent, et on affiche les données. On incrémente alors la variable qui compte le nombre d'affichages. Quand on arrive en bout de liste on s'arrête.

```c
int affichageReservationsPourUnJeu(ListeReservation lr, int idJeux, Adherent **tAdh, int nbAdh
{
    int i = 0, erreur;
    Adherent adh;

    if (lr == NULL)
        return 0;

    if (lr->r.idJeux == idJeux)
    {
        erreur = rechercheIdentiteAdherent(tAdh, nbAdh, lr->r.idAdherent, &adh);
        //Ne devrait JAMAIS arriver, uniquement si on supprime un adhérent alors qu'il possède
        if (erreur == -1)
            return affichageReservationsPourUnJeu(lr->suivant, idJeux, tAdh, nbAdh) + i;

        printf("%s\t%-30s\t%-30s\t%02d/%02d/%04d\n", adh.civilite, adh.nom, adh.prenom,
            lr->r.dateReservation.jour, lr->r.dateReservation.mois, lr->r.dateReservation.anne
        i++;
    }
    return affichageReservationsPourUnJeu(lr->suivant, idJeux, tAdh, nbAdh) + i;
}
```

Affichages de toutes les réservations qui ont le même ID de jeu !

3.5 La création d'un emprunt/réservation et/ou la création d'un adhérent

Une fois les affichages terminées, il fallait s'atteler aux modifications des données (suppressions, ajouts, changements).

On a commencé par faire l'ajout d'emprunt / réservations (fonctionnalité 5).

Pour cela nous avons commencé par faire une fonction qui vérifie si une inscription est toujours valide. Pour cela on envoie la date actuelle et la date d'inscription et on fait plusieurs tests pour vérifier que la date d'inscription a moins d'un an, si c'est le cas on retourne 0, 1 sinon.

On a fait une fonction qui récupère la date actuelle. Plutôt qu'une simple entrée au clavier, on trouvais cela plus intéressant de la récupérer avec la commande *system*, cela faisait écho à nos cours de systèmes de la P1.

```c
Date recupererDateActuelle(void)
{
    FILE *flot;
    Date d;
    system("date +%d/%m/%Y > date.txt");
    flot = fopen("date.txt", "r");
    fscanf(flot, "%d/%d/%d", &d.jour, &d.mois, &d.annee);
    fclose(flot);
    return d;
}
```

On récupère la date avec la commande systeme, *puis on l'écrit dans un fichier texte. On ouvre le fichier, on a lit, et on la retourne dans une structure Date.*

On a également fait une fonction qui recherche un ID d'emprunt disponible.

On a fait une fonction qui vérifie si une date d'emprunt est toujours valide, c'est à dire, a t-elle plus d'un mois ? (voir le programme pour plus d'infos).

Dans la fonction principale, on demande d'abord si l'adhérent est nouveau. S'il répond oui, on appelle la fonction qui ajoute un adhérent : on fait alors diverses entrées sécurisées, et on crée l'adhérent en l'ajoutant au tableau de pointeurs, on incrémente la taille logique.

S'il n'est pas nouveau, il se connecte avec son id et nom pour que ce soit sécurisé. Ensuite, on vérifie que son inscription n'a pas expirée (qu'il peut toujours emprunter). Si c'est pas le cas, il peut se ré-inscrire. Après il entre le nom du jeu qu'il doit emprunter et on appelle la fonction qui l'ajoute (elle crée un ID d'emprunt, ajoute l'emprunt avec toutes les données nécessaires, incrémente la taille logique).

Si le jeu n'est pas disponible (stock nul), alors on lui demande s'il veut l'emprunter. S'il dit oui, on rappelle notre fonction d'ajout de réservation, afin de l'ajouter dans la liste.

3.6 Le retour d'un jeu

Dans le cas où l'on retourne un jeu, on doit supprimer un emprunt mais ce n'est pas tout ! Il faut parcourir la liste des réservations afin de trouver (éventuellement) une réservation sur ce jeu, et alors la transformer en emprunt et dans le cas où on a plusieurs réservations sur ce jeu, il faut prendre la plus ancienne !

On a donc commencé par faire une fonction de recherche qui vérifie si une emprunt existe. On a fait une fonction dichotomique et qui retourne -1 si l'ID d'emprunt donné en paramètre n'est pas trouvé, la position de l'emprunt dans le tableau sinon. On récupère aussi les données de l'emprunt. Voici la fonctions :

```c
int rechercheEmprunt(Emprunt **tEmp, int nbEmp, int idEmp, Emprunt *emp)
{
    int debut = 0, milieu, fin = nbEmp - 1;

    while (debut <= fin)
    {
        milieu = (debut + fin) / 2;

        if (tEmp[milieu]->idEmprunt == idEmp)
        {
            *emp = tEmp[milieu];
            printf("%d %d %d", emp->idEmprunt, emp->idJeu, emp->idAdherent);
            return milieu;
        }

        if (tEmp[milieu]->idEmprunt < idEmp)
            debut = milieu + 1;
        else
            fin = milieu - 1;
    }
    return -1;
}
```

Fonction de recherche dichotomique à partir d'un ID d'emprunt !

Ensuite on a réalisé une fonction qui compare deux dates. Elle retourne 0 si la date 1 est plus grande que la date 2, 2 si elles sont égales, 1 sinon :

```c
int comparaisonDates(Date date1, Date date2)
{
    if (date1.annee > date2.annee)
        return 0;
    else if(date2.annee > date1.annee)
        return 1;

    if (date1.mois > date2.mois)
        return 0;
    else if(date2.mois > date1.mois)
        return 1;

    if (date1.jour > date2.jour)
        return 0;
    else if(date2.jour > date1.jour)
        return 1;

    return 2;
}
```

Compare deux dates. Utilisé pour trouver une réservation plus ancienne.

On a réalisé la fonction qui supprime un emprunt. Pour ce faire, on libère la mémoire de la position où se trouve la réservation qu'on veut supprimer, on parcours le tableau et on réalise un décalage des données, on décrémente la taille logique :

```c
void suppressionEmprunt(Emprunt **tEmp, int *nbEmp, int indice)
{
    int i;

    free(tEmp[indice]);

    if (*nbEmp != 1)
        for (i = indice; i <= *nbEmp - 2; i++)
            tEmp[i] = tEmp[i + 1];

    *nbEmp = *nbEmp - 1;
}
```

Suppression d'un emprunt.

On a réalisé la fonction principale du menu dans laquelle on demande à l'utilisateur d'entrer un id d'emprunt, on lui informe si l'ID existe ou non, on récupères les informations sur l'utilisateur de l'emprunt et sur le jeu qu'il a emprunté, et on vérifie qu'il s'agit bien de la bonne personne en lui demandant de rentrer son nom. Si l'ID de l'emprunt et le nom de la personne correspondent c'est que c'est bien sécurisé, on affiche le récapitulatif, on demande confirmation et s'il accepte, on supprime l'emprunt en appelant la fonction de suppression. On augmente le stock du jeu.

Si le jeu avait un stock nul, cela signifie qu'il y avait peut être des gens qui avaient fait des réservations ! Il faut donc vérifier s'il y en a eu ou pas, et pour cela on appelle une fonction chargée de transformer une réservation en emprunt.

Dans cette fonction, on récupère la date du jour avec la fonction correspondante, et on crée une date très élevée (année 30 000 par exemple), afin que lors des tests des dates, cela prenne bien la réservation la plus ancienne.

On appelle dans cette fonction la fonction qui va rechercher si une réservation est disponible :

```c
void rechercherReservationPlusAncienne(ListeReservation lr, Emprunt **tEmp, int nbEmp, Date *dateResa, Rese
{
    if (lr == NULL)
        return;

    if (lr->r.idJeux == idJeux && comparaisonDates(*dateResa, lr->r.dateReservation) == 0)
        if (compteNbEmp(tEmp, nbEmp, lr->r.idAdherent) < 3 && rechercheDateEmpruntDepasse(tEmp, nbEmp, lr->
        {
            *reserv = lr->r;
            *dateResa = lr->r.dateReservation;
        }

    rechercherReservationPlusAncienne(lr->suivant, tEmp, nbEmp, dateResa, reserv, idJeux, dateActu);
}
```

Cette fonction récursive parcours les réservations et se termine lorsqu'on arrive au bout de la liste. Si jamais pleins de critères différents sont respectés (voir le programme pour plus de détails), comme par exemple on vérifie que l'ID du jeu de la réservation corresponde au même que celui de l'emprunt, que cette date est bien la plus ancienne, que le nombre d'emprunt de la personne est inférieur à 3, que la personne a bien rendue tous ses emprunts à temps, alors on récupère la date de l'emprunt et les informations de cette réservations.

De retour dans l'autre fonction, on va vérifier que le mois récupéré n'est pas égal au mois initialisé de la fausse date qu'on a entré au début (on avait mis le mois 13, alors qu'il n'existe pas, donc il est forcément différent si on a trouvé une réservation sur ce jeu).

A partir de ce moment là, on diminue le stock, on appelle la fonction pour supprimer la réservation, et on appelle la fonction qu'on a crée qui ajoute un emprunt.

3.7 La suppression d'une réservation

Pour supprimer une réservation, nous avons tout d'abord crée la fonction qui va supprimer la réservation.

Nous avons du en créer deux. Une pour supprimer une valeur en tête de liste, et une autre pour choisir laquelle supprimer.

Celle qui supprime en tête est de fonctionnement simple. On crée une autre liste, on la fait pointer sur le maillon suivant de la liste originelle, on libère la mémoire du maillon actuel de l'autre liste, et on retourne la nouvelle.

```
ListeReservation suppressionEnTete(ListeReservation lr)
{
    Maillon *svt;
    svt = lr->suivant;
    free(lr);
    return svt;
}
```

On supprime l'élément en tête !

L'autre fonction récursive s'occupe d'appeler cette fonction de suppression si elle détecte que l'ID de la réservation du maillon actuel est le même que celui fournit en paramètre. Comme les ID sont uniques, si cette condition est vérifiée c'est qu'on supprimer bien la bonne !

```
ListeReservation suppressionDuneReservation(ListeReservation lr, int idResa)
{
    if (lr == NULL)
        return lr;

    if (lr->r.idResa == idResa)
    {
        return suppressionEnTete(lr);
    }
    if (lr->r.idResa > idResa)
        return lr;

    lr->suivant = suppressionDuneReservation(lr->suivant, idResa);
    return lr;
}
```

Fonction récursive qui appelle la fonction de suppression en cas de détection d'ID correcte.

Nous avons crée une fonction récursive qui recherche si un ID de réservation existe et copie dans une variable de type Réservation, la totalité des données de la réservation :

```c
int rechercheInfoResa(ListeReservation lr, int idResa, Reservation *reserv)
{
    if (lr == NULL)
        return 0;

    if (lr->r.idResa == idResa)
    {
        reserv->idJeux = lr->r.idJeux;
        reserv->idAdherent = lr->r.idAdherent;
        reserv->dateReservation.jour = lr->r.dateReservation.jour;
        reserv->dateReservation.mois = lr->r.dateReservation.mois;
        reserv->dateReservation.annee = lr->r.dateReservation.annee;
        return 1;
    }

    return rechercheInfoResa(lr->suivant, idResa, reserv);
}
```

Une fonction de recherche d'informations pour un ID de réservation !

Par la suite, nous avons créé la fonction principale de la suppression, celle qui va s'occuper de demander l'ID de la réservation ainsi que le nom de la personne (sécuriser les réservations des gens, pour éviter que n'importe qui puisse supprimer n'importe quoi).

On vérifie sur l'ID de réservation entré correspond avec le nom de la personne.

Si tout fonctionne bien, on affiche les données de la réservation et on demande la confirmation de la suppression… Si oui, alors on supprime en appelant la fonction de suppression. On oublie pas de retourner à la fin la liste de réservation qui a peut-être été modifiée.

Comme la plupart des fonctions appelées par la menu, cette fonction ne fait que des l'affichage / des entrées / des appels à des fonctions de recherche.

3.8 La sauvegarde des données

Nous devions enregistrer les données modifiées sans quoi tout ce travail était inutile.

Pour la liste chaînée il suffisait de sauvegarder les données dans le même fichier que celui où on les a chargé :

```c
int enregistrementReservation(ListeReservation lr)
{
    FILE *fs;
    fs = fopen("reservations.don", "w");
    if (fs == NULL)
    {
        printf("Problème lors de l'enregistrement des données
        return -1;
    }

    ecritureFichierReservation(lr, fs);

    fclose(fs);
    return 0;
}
```

On ouvre le fichier, et on écrit les données dedans. La fonction d'écriture est une fonction récursive qui inscrit toutes les données de la liste chaînée dans le fichier, jusqu'à arriver au bout.

Pour les tableaux de pointeurs d'adhérents, de jeux, et d'emprunts, il nous suffisait de faire une fonction de sauvegarde très similaire, sauf qu'on utilise une boucle *for* pour parcourir tout le tableau et à chaque itération, on utilise *fwrite* pour inscrire les données du tableau dans le fichier binaire.

Pas besoin de faire de fonction pour sauvegarder les données dans un fichier régulier ! Il sert juste au départ, à charger les données une première fois.

IV – La répartition du travail

Durant ce projet nous nous sommes répartis les tâches de manière à ce que chaque personne travaille avec ses données qu'il aura chargé. Voici donc comment s'est déroulé la répartition des tâches :

Concernant les fonctionnalités 1 et 8 (chargement / sauvegarde) :
-Adrien a réalisé le chargement et la sauvegarde des données des adhérents, des emprunts (chargement des fichiers réguliers et enregistrement des données (lecture) dans des tableaux de pointeurs, sauvegarde des données dans des fichiers binaires, chargement des données depuis un fichier binaire, fonction d'affichage de débogage).
-Maxime a fait le chargement et la sauvegarde des données des jeux (chargement du fichier régulier et enregistrement des données (lecture) dans un tableau de pointeurs, sauvegarde des données dans un fichier binaire, chargement des données depuis un fichier binaire, fonction d'affichage de débogage).
-Jérémy a fait le chargement des réservations et leur enregistrement dans le même fichier (chargement depuis fichier régulier, sauvegarde dans un fichier régulier, lecture + écriture, comprend les fonctions d'insertions, de création de liste vide, et d'affichage de débogage). Il a réalisé la fonction de tri rapide des emprunts par id, puis le tri fusion par date d'emprunt (pour la sauvegarde).

Concernant les fonctionnalités 2, 3, 4 (affichages) :
-Adrien a réalisé l'affichage des emprunts, a réalisé une fonction de recherche pour les adhérents, pour les jeux.

-Maxime a réalisé le tri des jeux par ordre alphabétique du nom, ainsi que leur affichage.
-Jérémy a réalisé une fonction de recherche pour les jeux, une fonction récursive d'affichage, une fonction de saisie de jeu et d'affichage de ces emprunts.

Concernant la fonctionnalité 5 :
-Maxime a réalisé la connexion d'un adhérent, son inscription dans le cas où il est nouveau (fonction d'insertion). Il a fait l'ajout de l'emprunt (décalage + insertion). Il a réalisé une fonction qui vérifie si une date d'inscription d'un adhérent et toujours valide, le cas échéant, la modification de son inscription.
-Jérémy a fait les tests pour vérifier que l'emprunt était possible (fonction qui compte le nombre d'emprunts, fonction qui vérifie si un emprunt a plus d'un mois, fonction qui recherche un id d'emprunt disponible…)

Concernant la fonctionnalité 6 :
-Adrien a réalisé la fonction principale du retour d'un jeu. Il a réalisé la suppression d'un emprunt (décalage).
-Jérémy a réalisé la fonction qui transforme un emprunt en réservation, ainsi que celle qui parcours les réservation afin de sélectionner une réservation la plus ancienne.

Concernant la fonctionnalité 7 :
-Jérémy a réalisé les fonctions de suppression EnTete et suppression d'une réservation, il a aussi fait la fonction principale ou l'adhérent se « connecte » (rentre son id, son nom), une fonction de recherche d'une réservation.

Concernant les fonctions de récupération de date, de comparaison, les commentaires, les menus, les fonctions de tests… :
-On a travaillé tous ensemble !