Lefebvre Clémence

Laviolette Etienne

NF16 – Compte rendu TP03

Représentation et manipulation d'une ludothèque

Pour le vendredi 21 Novembre 2014

utc
Université de Technologie
Compiègne

Introduction :

Le but de ce TP numéro trois de NF16 est de développer une application console simplifiée dans le but de gérer une ludothèque à l'aide de listes chaînées. Il constitue donc une application directe en langage C des algorithmes vus et expérimentés lors des cours et des travaux dirigés depuis le début de l'année dans l'uv.

Notre objectif lors de la réalisation de ce TP fut de créer un programme correct, robuste ainsi que le plus simple possible de compréhension et d'utilisation.

Dans ce rapport nous allons vous présenter tout d'abord les structures utilisées tout au long du développement et les choix qui ont été les nôtres, puis nous aborderons les fonctions qui ont été réalisées, leur complexité et les choix que nous avons faits au cours de la réalisation du travail demandé.

I. Présentation du problème et fonctions à réaliser :

Afin de gérer notre ludothèque nous avons suivis les préconisations de l'énoncé quand à la définition du problème.

Il s'est agi lors de ce TP de gérer une ludothèque et les jeux qui la composent.

Pour ce faire nous avons repris les informations de l'énoncé concernant la constitution d'un jeu :

- Son nom,
- Son genre,
- Le nombre de joueurs minimum pour ce jeu
- Le nombre de joueurs maximum pour ce jeu
- La durée moyenne d'une partie de ce jeu en minutes ;

Secondement une ludothèque doit contenir :

- Le nombre de jeux qu'elle contient
- Les jeux ;

Afin de répondre le plus précisément à ces spécifications techniques nous avons décidé d'introduire les structures suivantes.

II. Structures, types et cadre de développement :

Dans le but de développer le plus aisément possible notre programme nous avons repris les structures présentes dans l'énoncé puis nous avons introduit celles qui nous semblaient les plus pertinente. Ce travail a d'abord été réalisé à la main et cette phase de recherche préalable au codage à proprement parlé de notre programme fut le moment le plus important puisqu'il a décidé de notre orientation.

Dans un souci de clarté, toutes les définitions de structures et de types ont été répertoriées dans le fichier TP3.h qui constitue le seul header de notre programme.

Tout d'abord le premier niveau d'abstraction en langage C a été de formaliser et de définir le genre d'un jeu. Nous l'avons fait grâce au type énumération « genre_jeu » :

Toute variable de type « genre_jeu » peut prendre une des valeurs possibles suivantes :

0 : Plateau,

1 : RPG,

2 : Coopératif,

3 : Ambiance,

4 : Hasard

```
enum genre_jeu
{
    PLATEAU, RPG, COOPERATIF, AMBIANCE, HASARD
};
typedef enum genre_jeu genre_jeu;
```

Pour manipuler un jeu dans le programme C nous avons défini la structure suivante :

Cette structure « jeu » contient plusieurs champs :

- Un champ « nom » qui est une chaîne de caractère et contiendra le nom du jeu,
- Un autre « genre » de type genre_jeu défini ci-dessus.
- Un champ entier « nbJoueurMin » le nombre de joueurs minimum pour jouer à ce jeu
- Un seconde champ entier « nbJoueurMax » définissant le nombre de joueurs maximum pour ce jeu
- Un champ « duree » de type entier qui contiendra la durée moyenne d'une partie
- Un pointeur « suivant » sur le jeu suivant dans la ludothèque

```
struct jeu
{
    char* nom;
    genre_jeu genre;
    int nbJoueurMin;
    int nbJoueurMax;
    int duree;
    struct jeu* suivant;
};
```

Afin de simplifier nos déclarations de type nous avons associé à cette structure le type t_jeu défini ci-dessous :

```
typedef struct jeu t_jeu;
```

Pour représenter une ludothèque dans notre programme nous avons introduit la structure « ludotheque ». Elle se compose des champs suivants :

- Un entier « nb_jeu » contenant le nombre de jeu de cette ludothèque
- Un pointeur nommé debut qui pointe sur le premier jeu de cette ludothèque

```
struct ludotheque
{
    int nb_jeu;
    t_jeu * debut;
};
```

Comme pour les jeux nous lui avons associé le type t_ludotheque par un typedef :

```
typedef struct ludotheque t_ludotheque;
```

Lors de la phase de conception de ce programme nous avons décidé d'introduire d'autres notions qui, nous avons pensé, nous aideraient à manipuler les ludotheques.

Tout d'abord nous avons créé un tableau de pointeur de ludothèques nommé « tab_ludo ». Indicé par a, nous avons décidé de créer un programme qui contiendrait au maximum 10 ludothèques différentes.

```
t_ludotheque* tab_ludo[10];
for (a=0;a<10;a++)
    tab_ludo[a]=NULL;
```

De la même manière nous avons répertorié tous nos jeux dans un tableau de jeu nommé « tab_jeux » qui est limité à 20 jeux différents. Ce tableau contient les adresses de tous les jeux stockés en mémoire à l'instant t.

```
t_jeu* tab_jeux[20];
for (a=0;a<20;a++)
    tab_jeux[a]=NULL;
```

<u>Remarque :</u> L'utilité et la pertinence a posteriori seront étudiés dans une partie ultérieure de ce rapport.

Le type central découlant des définitions ci-dessus ayant été utilisés lors du développement et notamment lors du découpage du programme en fonction est le suivant.

t_ludotheque * comme pointeur vers une ludothèque :

```
t_ludotheque* ludo1;
```

III. Description des fonctions de notre programme :

Dans le désir d'une clarté optimale de notre fonction main () nous avons essayé notre programme en différentes sous fonctions. En voici la description brève.

a.) Création d'une ludothèque :

Cette fonction est définie comme ceci :

Elle renvoie donc un pointeur vers une ludothèque. Ce pointeur renvoyé nommé « ludo_vide » est créé et un espace lui est réservé grâce à un malloc. Si cette allocation réussit nous initialisation les champs de cette ludothèque vide créée :

Le nombre de jeu à 0 et le pointeur vers le premier jeu à NULL. Notre ludothèque est donc vide pour le moment.

```
t_ludotheque * creer_ludotheque(){
  t_ludotheque* ludo_vide = malloc(sizeof (t_ludotheque));

  if (ludo_vide){
        ludo_vide->nb_jeu = 0;
        ludo_vide->debut = NULL;
        return ludo_vide;
  }
  else
    return NULL;
}
```

Nous retournons enfin notre « ludo_vide » pour finir cette fonction.

b.) Création d'un jeu :

Voici le prototype de la fonction « créer_jeu » qui renvoie un pointeur vers le jeu créé et prend en arguments tous les constituants du jeu définis dans la partie II de ce rapport.

Pour réaliser cette fonction nous créons un « new_jeu » de type t_jeu * grâce à un malloc et demandons à l'utilisateur de rentrer au clavier les différents éléments champs du jeu en question. Nous les affectons à « new_jeu » grâce à l'opérateur « -> » puisque le « new_jeu » est de type pointeur vers un t_jeu.

Si le malloc a échoué, aucune information n'est demandée.

```
t_jeu * creer_jeu(char * nom, enum genre_jeu genre, int nbJoueurMin, int nbJoueurMax, int duree){
    t_jeu * new_jeu = malloc(sizeof(t_jeu));
    if(new_jeu)
    {
        char* name = malloc(20* sizeof(char));
        strcpy(name,nom);
        new_jeu->nom    = name;
        new_jeu->genre = genre;
        new_jeu->nbJoueurMin = nbJoueurMin;
        new_jeu->nbJoueurMax = nbJoueurMax;
        new_jeu->duree = duree;
        new_jeu->suivant=NULL;
        return new_jeu;
    }
    else
        return NULL;
}
```

c.) Ajout d'un jeu :

Une fois la ludothèque et le jeu créés, il nous fallait coder la fonction d'ajout d'un jeu à une ludothèque. C'est le rôle de la fonction « ajouter_jeu » qui prend en argument deux pointeurs : l'un vers la ludothèque dans laquelle ajouter le jeu et l'autre vers le jeu en question.

```
int ajouter_jeu(t_ludotheque* ludo, t_jeu* j){

    if(!ludo){
       printf("Cette ludotheque n'existe pas\n");
       return 0;
       }
    t_jeu* jeu_temp = ludo->debut;
    if ((jeu_temp==NULL)||(strcmp((j->nom),(jeu_temp->nom))<0)){ //si j inférieur à premier ou ludo vide
          j->suivant=ludo->debut;
          ludo->debut=j;
          ludo->nb_jeu++;
          return 1;
       }

       else{
          while ((jeu_temp->suivant)!=NULL && strcmp((j->nom),(jeu_temp->suivant->nom))>0){
              jeu_temp=jeu_temp->suivant;
          }
          j->suivant =jeu_temp->suivant;
          jeu_temp->suivant = j;
          ludo->nb_jeu++;
          return 1;
       }
}
```

Cette fonction se compose d'une structure conditionnelle « Si.. Sinon » afin d'exclure immédiatement le cas où la ludothèque dans laquelle on demande l'ajout du jeu n'a pas été créée au préalable.

Au cas où elle existe nous définissions un « jeu_temp » (jeu temporaire) qui est un pointeur initialisé vers le premier jeu de la ludothèque passée en paramètre. Il va nous servir à parcourir tous les jeux de la ludothèque les uns à la suite des autres afin d'ajouter le jeu à sa place selon l'ordre alphabétique.

C'est l'objectif du « while » de ce ce « else » qui a deux conditions : soit on vient de trouver la place du jeu à ajouter, soit on l'ajoute en dernier. Afin de savoir où ajouter le jeu nous avons utilisé la fonction strcmp qui manipule et compare selon l'ordre lexicographique des deux chaines de caractère passées en arguments. Cette fonction renvoie 0 si les chaines sont identiques, un nombre positif si la première est après la seconde et un nombre négatif sinon.

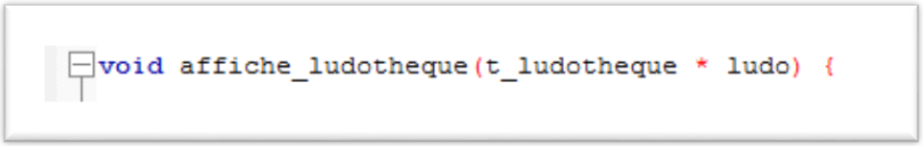
Lorsque nous avons trouvé la place du jeu dans la ludothèque nous établissons les liens avec le jeu suivant le jeu précédent pour garder l'intégrité de la liste chaînée. Une incrémentation du nombre de jeux de la ludothèque s'effectue en même temps pour que les champs et de la ludothèque coïncident avec son contenu.

La fonction est de type int puisqu'elle renvoie 0 si la ludothèque passée en paramètre n'existe pas ou 1 dans les autres cas.

Remarque : L'utilisation de la fonction strcmp a engendré l'inclusion en en-tête de la bibliothèque string.h.

d.) L'affichage d'une ludothèque :

Les trois fonctions précédentes nous ont permis de créer et remplir des ludothèques avec des jeux. La fonction affiche_ludotheque prenant un pointeur vers une ludothèque permet d'afficher à l'écran son contenu.



Un test est effectué pour savoir si cette ludothèque existe ou non. Puis un jeu temporaire de type t_jeu * permet de parcourir l'ensemble des jeux de la ludothèque passée en paramètre et de les affiche selon un format défini. Une étude de la largeur graphique de la console couplée à une manipulation des possibilités de la fonction printf nous a permis de définir un standard d'affichage aligné à gauche des jeux d'une ludothèque.

La boucle while testant si le jeu courant existe ou si nous sommes arrivés à la fin du parcours des jeux permet de s'arrêter au bon moment.

Cette fonction est de type void puisque son seul but est l'affichage.

e.) Le retrait d'un jeu :

Après avoir créé les ludothèques, leurs jeux et l'affiche de celles-ci, la suite logique dans la gestion est la suppression d'un jeu.

Pour ce faire nous avons décidé d'écrire la fonction de type int suivante :

```
int retirer_jeu(t_ludotheque* ludo, char* nom) {
```

Les paramètres de cette fonction sont un pointeur vers la ludothèque dans lequel se situe le jeu et son nom.

La structure de cette fonction est le suivante : Si.. Sinon Si.. Sinon

Le premier « si » test si la ludothèque est vide ou n'existe pas. Dans les deux cas le retrait échoue et la fonction renvoie 0.

Le « sinon si » retire le premier jeu de la ludothèque si celle-ci n'en contient qu'un.

Enfin dans le « sinon » nous allons chercher le jeu dans la ludothèque puisque celle-ci existe, et est non vide et contient plus d'un jeu. Grâce à un t_jeu * temporaire initialisé sur le premier jeu de la ludothèque nous allons la parcourir en testant individuellement le premier jeu puis dans un « while » les suivants. La même fonction strcmp qu'utilisée précédemment testera la concordance entre deux chaines de caractères.

Lorsque nous avons trouvé le jeu à retirer, nous chaînons le jeu précédent celui-ci avec celui le succédant puisque nous libérons l'espace mémoire occupé par ce jeu à l'aide d'un free(supp).

L'idée principale a été de toujours garder, à chaque moment de la boucle, le jeu précédent et le jeu suivant afin de rétablir le chainage. Cela a été réalisé en testant la condition sur le jeu suivant (jeu_temp) tout en connaissant le jeu actuel (mémoire). Nous retournons 1 lorsque la suppression s'est bien déroulée et 0 si le jeu n'existe pas.

f.) Suppression d'une ludothèque :

La fonction de suppression d'une ludothèque est de type void puisqu'elle ne sert qu'à la manipulation interne de données dans le programme.

Voici son prototype :

```
void supprimer_ludotheque(t_ludotheque * ludo){
    if (!ludo)
        printf("cette ludotheque n'existe pas...\n")
    else{
        t_jeu* jeu_temp = ludo->debut;
        t_jeu* memoire;

            while(jeu_temp!=NULL){
                memoire=jeu_temp;
                jeu_temp=jeu_temp->suivant;
                free(memoire);
            }
    }
    free(ludo);
}
```

Comme décrit dans son prototype, la fonction de suppression ne prend qu'un seul paramètre : un pointeur vers la ludothèque à supprimer.

Son fonctionnement est assez simple : une structure « Si..Sinon » qui test tout d'abord si la ludothèque passée en paramètre existe ou pas (si elle a été créée avant ou pas).

Dans le cas contraire deux pointeurs de jeu sont créés. L'un sert à parcourir les jeux tandis que l'autre stock tour à tour tous les jeux à effacer. La mémoire pris par chaque jeu est libérée à l'aide d'un free. Une fois la ludothèque vidée. On peut libérer l'espace mémoire occupé par la ludothèque elle-même.

g.) Recherche d'un ou plusieurs jeux :

Une interaction importante avec l'utilisateur est la recherche d'un ou plusieurs jeux correspondants à un ou plusieurs critères dans une ludothèque donnée.

Cette fonction remplit ce rôle :

```
t_ludotheque * requete_jeu(t_ludotheque * ludo, enum genre_jeu genre, int nbJoueurs, int duree) {
```

Cette fonction de recherche de jeux prend comme paramètres : la ludothèque dans laquelle effectuer la recherche, un genre, un nombre de joueurs et une durée.

Nous allons utiliser un t_jeu * pour parcourir l'ensemble des jeux de la ludothèque, un pointeur vers une ludothèque nommé « jeux_possibles ».

En premier lieu nous copions intégralement la ludothèque que nous allons filtrer avec des « si successifs » qui testent à chaque fois si le jeu actuellement parcouru correspond à un critère de genre, de nombre de joueurs ou de durée. Si ça n'est pas le cas on le supprime de la ludothèque « copiée ».

Ainsi à la fin du filtrage on se retrouver avec une ludothèque qui contient seulement les jeux correspondants à la requête.

Nous pouvons renvoyer le contenu de cette ludothèque « jeux possibles ».

Remarque : Pour calculer la durée avec un écart de +- 10% nous avons introduit la variable locale « écart » qui calcule l'écart avec la durée entrée par l'utilisateur et test si le jeu actuellement parcouru est dans cet écart.

h.) La fusion de deux ludothèques

La fonction suivant réalise la fusion de deux ludothèques :

```
t_ludotheque* fusion_deux_ludo(t_ludotheque * ludo1, t_ludotheque * ludo2)
```

Ludo_1 et ludo_2 seront ces deux ludothèques quant aux deux pointeurs vers des t_jeux, ils permettront encore une fois de maitriser le jeu actuellement parcouru et le suivant.

Nous testons tout d'abord si la première ludothèque est vide ou n'existe pas -> dans ce cas la fusion renvoie la seconde.

De même avec la seconde ludothèque.

Nous avons traité les cas spéciaux.

Nous copions l'intégralité des ludothèques à copier grâce à une fonction de service introduite plus loin dans ce rapport.

La « new_ludo », copie de la première sera renvoyée. Elle sert de base à la comparaison des jeux pour éliminer les doubler et sert aussi de support à l'ajout des jeux de la seconde ludothèque.

Un pointeur itératif de type t_jeu * va parcourir l'ensemble des jeux de la seconde ludothèque et les ajouter grâce à la fonction ajouter_jeu dans la copie de la première : dans « new_ludo ».

Une fois cette copie effectuée (avec conservation de l'ordre lexicographique puisque la fonction ajouter_jeu ajoute par ordre), nous allons parcourir cette « new_ludo » pour éliminer les doublons éventuels grâce à un « while ».

Nous avons défini un doublon comme étant un jeu ayant toutes ses caractéristiques similaires à un autre déjà présent : le même nom, les même nombres de joueurs max et min, la même durée, le même genre.

Pour finir nous affichons cette « new_ludo » grâce à la fonction afficher_ludotheque puis nous renvoyons le pointeur vers celle-ci.

i.) Fonctions de service :

- La fonction copier_ludo prend en paramètre un pointeur vers la ludothèque à copier et renvoie une adresse vers la copie de la ludothèque :

```
t_ludotheque * copier_ludo (t_ludotheque * ludo){
```

Nous allouons donc l'espace pour une ludothèque grâce à un malloc.

Nous utilisons un t_jeu * pour parcourir la ludotheque à copier et allouons dans une boucle « while » de l'espace pour chaque jeu appelé « new_jeu » de la ludothèque copie nommée « new_ludo ».

Il a fallu allouer par un malloc de l'espace pour stocker le nom récupéré par strcpy s sinon la copie du nom ne s'effectuait pas.

Remarque : Cette fonction a été ajoutée à notre programme afin de résoudre le problème rencontré lors de la démo. En effet sans une copie de la liste chainée nos fonctions modifiait le chainage de nos ludothèques pour obtenir le résultat souhaité ce qui empêchait de réutiliser ensuite la ludothèque originale. Grâce à cette fonction nous pouvons réaliser une copie de la ludothèque originale avant toute modification."

- La fonction liste_ludotheque permet de liste à l'écran les ludothèques existantes :

```
int liste_ludotheque(t_ludotheque* tab_ludo[10]){
```

Elle parcourt le tableau de ludothèque et renvoie le numéro de celle qui sont initialisées (case du tableau d'adresses de ludothèques différent de NULL). La boucle for a été utilisée par rapport à la while pour gérer toutes les situations : notamment celle où on supprimait la ludothèque numéro 1 alors que la 2 existe toujours.. Avec un while nous ne pouvons plus y avoir accès. Cette fonction retourne la nombre de ludothèques dans le tableau

- La fonction nombre_ludotheque renvoie le nombre de ludothèques initialisées dans le tableau de ludothèque sans les afficher (contrairement à la fonction précédente) :

```
int nombre_ludotheque (t_ludotheque* tab_ludo[10]){


        int i;
        int nbre= 0;

        for (i=0; i< 10; i++){
            if (tab_ludo[i] != NULL)
                    nbre++;
                }
        return nbre;
}
```

Comme précédemment cette fonction est réalisée à l'aide d'un for.

- La fonction « indice_jeu » renvoie la première case vide du tableau de jeux. Elle parcourt ce tableau à l'aide d'un for et renvoie le premier indice dont la case est égale à NULL et renvoie l'indice en question :

```
int indice_jeu (t_jeu* tab_jeux[20]){
        int i = 0;

            while (tab_jeux[i]!=NULL){
                i++;
            }
        return i;

}
```

- La fonction « premier_case_vide » fait la même chose avec le tableau des ludothèques en renvoyant cet indice.

```
int premiere_case_vide (t_ludotheque* tab_ludo[10]){
```

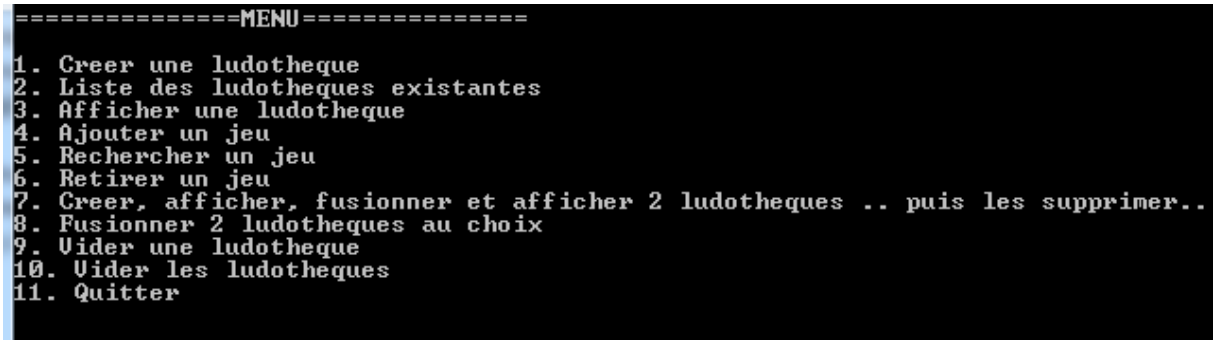
IV. Fonction principale main () :

Afin de gérer le plus simplement les interactions avec l'utilisateur nous avons découpé notre programme en diverses sous fonctions détaillées ci-avant. Cependant l'importance de la fonction principale main reste très importante et il nous a semblé bon d'en détailler précisément son fonctionnement.

La structure globale qui a été choisie est un while sur une variable nommé « fin ». Si cette variable passe à 1 (dans le cas où l'utilisateur rentre le choix 11 de notre menu console) alors le programme se termine.

Nous y avons ajouté quelques fantaisies graphiques afin de rendre le programme plus personnel. Celles-ci ne seront pas abordées ici.

Notre menu de gestion de ludothèque est le suivant :



Chaque choix est géré par un case sur la variable « choix ».

Case n°1 : Création d'une ludothèque avec utilisation de la fonction créer_ludothèque() qui ne prend aucun paramètre. Puis teste de la valeur de retour et affichage à l'écran selon celle-ci.

Case n°2 : Affichage des ludothèques existantes en utilisant les fonction nombre_ludotheque et liste_ludotheque avec comme paramètre notre « tab_ludo » qui est l'identificateur du tableau de ludothèques dans ce main(). Test de la valeur de retour de nombre_ludotheque et affichage en conséquent. (Notamment l'accord du verbe « exister » en fonction du retour d'une ou plusieurs ludothèques).

Case n°3 : Affiche des ludothèques disponibles (créées) grâce à nombre_ludotheque et liste_ludotheque. Récupération du choix de l'utilisateur dans « choix ». Test de ce choix dans le tableau des ludothèques (le choix rentré correspond à l'indice de la ludothèque en question dans le tab_ludo) pour vérifier que le choix est valide. Affichage de la ludothèque choisie si le choix est valide.

Case n°4 : Ajout d'un jeu dans une ludothèque.

Test si des ludothèques sont créées. Si oui récupérations des caractéristiques du jeu dans des variables locales. Récupération de la première case disponible dans le tableau des jeux grâce à indice_jeu puis appel de créer qui stockera le pointeur de retour à l'indice trouvé précédemment dans le tableau des jeux. Le test du choix valide ou non est le même que dans le case 3.

Case n°5 : Recherche d'un jeu dans une ludothèque.

Affichage des ludothèques disponibles, récupération du choix puis test de la validité de ce choix. Si ce choix est valide on vérifie si la ludothèque est vide. Si oui on renvoie un message d'erreur. Sinon on poursuit le processus en récupérant les critères puis en appelant la fonction requete_jeu à l'aide de ces critères et du tab_ludo[choiix]. Affichage de la ludothèque temporaire contenant les jeux correspondants aux critères puis suppression de celle-ci.

Case n°6 : Retrait d'un jeu.

On affiche les ludothèques disponibles puis on demande un choix dont on teste la validité. On vérifié ensuite si la ludothèque est vide : dans ce cas la recherche est impossible. Sinon on demande le nom du jeu à retirer, on appelle la fonction retirer_jeu avec le tableau de ludothèques et le nom du jeu. Puis on teste la valeur de retour de cette fonction pour générer l'affichage.

Case n°7 : Fusion de deux ludothèques rentrées par le programmeur.

Création des deux ludothèques grâce à la fonction créer_ludotheque. Insertion de chaque jeu grâce à la fonction ajouter_jeu. On appelle fusion_ludotheque dont on récupère la valeur de retour qui n'est autre que l'adresse de la ludothèque résultat de la fusion. On l'affiche grâce à affiche_ludotheque. On supprime ensuite les ludothèques et on réinitialise leur ancienne place dans le tableau de ludothèque à NULL.

Case n°8 : Fusion de deux ludothèques au choix.

Affichage des ludothèques disponibles grâce à nombre_ludotheque et liste_ludotheque. Récupération de deux choix « chx1 » et « chx2 ». Appel de ludo_fusion sur ces deux choix. Récupération du pointeur vers la ludothèque fusionnée (la valeur de retour de la fonction) puis insertion de celle-ci à la première case vide du tableau de ludothèques grâce à la fonction premiere_case_vide(). On finit par affiche le numéro dans la base de données que constitue le tableau de ludothèque à l'écran pour que l'utilisateur puisse s'en servir de nouveau ensuite.

Case n°9 : Vidage d'une ludothèque au choix.

On affiche les ludothèques disponibles comme précédemment. On demande un choix à l'utilisateur que l'on vérifie. S'il est valide on appelle la fonction supprimer_ludo sur tab_ludo[choiix] puis on réinitialise la case précédemment utilisée par cette ludothèque dans le tableau de ludothèque à NULL pour la utiliser de nouveau ensuite.

Case n°10 : Vidage de toutes les ludothèques.

On récupère le nombre de ludothèques existantes grâce à nombre_ludotheque. On parcourt le tableau de ludothèques en entier (entre 0 et 20 comme défini par le programmeur). Si la case est utilisée (différentes de NULL) on appelle supprimer_ludotheque sur cette case puis on réinitialise la case à NULL.

Case n°11 : Fermeture du programme.

Aucune interaction avec l'utilisateur simplement un vidage des toutes les ludothèques grâce au parcourt en entier du tableau de ludothèques. Puis affichage des crédits du programme.

V. Complexité de nos fonctions :

La complexité est un élément essentiel à prendre en compte lors du développement d'un programme et c'est un état des lieux du programme et de sa manière de fonctionner qu'il est indispensable de faire après coup. Nous allons donc détailler ici la complexité de chacune de nos fonctions détaillées dans la partie 4 de ce rapport.

- créer_ludotheque() : - Une affectation par malloc + une structure conditionnelle qui sera dans le pire des cas de deux affectations et un retour de fonction. Nous sommes dans le cas d'une complexité temporelle constante. O(1) dans le pire des cas Ω(1) dans le meilleur des cas donc une complexité exacte en θ(1).

- créer_jeu() :

  ➔ Dans le meilleur des cas : une affection par malloc puis un return. Donc Ω(1)

  ➔ Dans le pire des cas : une affectation par malloc puis une structure conditionnelle avec une nouvelle affectation par malloc puis l'appel à une fonction strcpy puis 6 affectations et un return. Notre fonction strcpy prend en paramètre deux adresses vers la premier caractère de chaque chaîne puis copie intégralement le deuxième dans la seconde. Ce sont donc des affectations successives du même nombre que la taille max de la chaîne « nom ». Nous l'avons borné à 20 donc au max, il y aura 20 affectations qui sont de l'ordre de 1 chacune.
  Donc O(1).

  ➔ Donc θ(1). Exactement

- ajouter_jeu(),affiche_ludotheque(),retirer_jeu(),supprimer_ludotheque(), copier_ludo() :

  ➔ Dans le meilleur des cas comme dans le pire on copie la ludothèque intégralement et on parcourt l'ensemble de ses jeux à l'aide du while. On parcourt de bout en bout la liste chaînée des jeux. Si nous avons n jeux :

  O(n)

  Ces fonctions s'apparentent (en complexité) au parcours complet d'une liste chaîné.

- fusion() :

  ➔ De l'ordre de 2* créer_ludo + 2*n puisqu'on parcourt une fois l'ensemble d'une liste pour la copier dans l'autre. Puis une deuxième fois pour gérer les doublons donc :
  O(n)

- requete_jeu()

  ➔ Cette fonction est de l'ordre de créer_ludo + 2*n donc :
  O(n)

- fusion_deux_ludo()

  ➔ Au vu des appels de cette fonction de l'ordre de 2*(copier_ludo) + n*(ajouter_jeu) donc en
  $O(n^2)$

- liste_ludotheque :

  ➔ Cette fois-ci on parcourt la liste des ludothèques. Si on considère que la taille de notre problème est m : nombre de ludothèque + nombre de cases remplies., cette fonction est donc en :
  O(m)

- nombre_ludotheques() :
  - ➔ Cette fonction est en O(p) avec p qui est la taille du taille des ludothèques.

- indice_jeu() :
  - ➔ En O(q) avec q qui est le nombre de jeux dans le tableau des jeux.

- premiere_case_vide() :
  - ➔ En O® avec r qui est le nombre de ludothèques à supprimer.

VI. Suggestions et critiques :

- Nous aurions pu encore plus découper notre programme en sous fonction. On remarque (et c'est encore plus marquant lors de la rédaction de ce rapport) que le main() fait de nombreuses fois les mêmes actions : On recueille un choix et on le test ou bien on affiche les ludotheques disponibles..

- Quelle est réellement l'utilité du tableau de jeux. On comprend clairement celui du tableau de ludothèques mais celui des jeux est-il utilisé souvent ?
  Il ne l'est qu'au moment de l'ajout d'un jeu dans une ludothèque mais cette étape est-elle nécessaire ? Au vu de notre programme actuel non. Nous aurions pu nous en passer.

- Des noms de variables plus explicites auraient pu être utilisés pour permettre une meilleure compréhension du programme par un tiers non développeur.
- Nous aurions pu renvoyer avec perror l'erreur possiblement issue de chaque malloc lors du test de la valeur de retour de celui-ci.

- La bibliothèque « ncurses » contenant de nombreuses fonctions d'affichage standardisée aurait pu être utilisée pour construire un affichage console plus simple d'utilisation et plus interactif.

- Le traitement des doublons aurait pu être géré d'une autre manière et éviter ainsi un second parcours en entier de nos listes chaînées.

- Nous aurions pu étudier la complexité spatiale notamment du fait des nombreux malloc de jeux ou de ludothèques.

- Une redondance entre les noms des variables du main() et du fichier TP3.c est à signaler. Cela peut perturber le lecteur sur la portée des différentes variables et leur valeur à un instant t (en dehors ou dans la fonction).

- Une retranscription algorithmique de chacune des fonctions a été évoquée mais jugée fastidieuse par les développeurs du programme.

- Certains choix (comme la définition d'un doublon) ont été pris mais sont à discuter puisqu'ils souffrent d'une carence en spécification précise.

- D'autres fonctionnalités auraient pu être ajoutées : ne pas arrêter totalement l'ajout d'un jeu si l'utilisateur se trompe au moment de rentrer le numéro de la ludothèque dans laquelle insérer le jeu. Proposer des choix de validation (être vous sur ?) avant

de lancer. Proposer de boucler l'ajout de plusieurs jeux à la suite. Ou bien encore proposer d'enregistrer la ludothèque créée lors de la recherche par de jeux par critères.

Conclusion :

Ce TP numéro 3 nous a permis de nous familiariser avec les manipulations de listes chaînées et de tous les algorithmes « type » qui les accompagne : recherche, insertion, suppression…

De plus nous avons pu développer des connaissances dans les manipulations de pointeur de pointeur (comme le tableau d'adresses de ludothèque), chose que nous n'avions pas pu expérimenter à l'occasion d'un projet d'une telle ampleur.

Aussi, la place primordiale de la réflexion en phase de conception et le travail en groupe sur un premier projet conséquent en NF16 en binôme est désormais acquis et non négligeable pour la suite de notre parcours.