TP5 : Structures

MP2I Lycée Pierre de Fermat

Consignes

Vous trouverez sur Cahier de Prépa une archive contenant des fichiers pour ce TP. La date limite de rendu est le ***dimanche 18/11 à 22h00***. N'oubliez pas de supprimer les exécutables, et de ranger vos exercices dans des dossiers.

N'oubliez pas de **commenter vos fonctions**, avec un commentaire de documentation au dessus de l'en-tête de la fonction, et des commentaires au sein du code pour expliquer son fonctionnement si besoin. Le but du commentaire de documentation est d' expliquer le rôle de la fonction, ce qu'elle fait, sans expliquer les rouages internes de son implémentation. On doit juste donner la spécification, c'est à dire décrire les sorties en fonction des entrées, donner les préconditions éventuelles, etc...

Il faut impérativement mentionner explicitement **le nom de chaque paramètre** de la fonction.

Par ailleurs, un commentaire de fonction doit être écrit pour être lu non pas par moi mais par n'importe qui qui lirait votre code indépendamment du fait que vous l'avez écrit pour un TP. Par exemple, n'écrivez pas `/* Cette fonction applique l'algorithme décrit à la question 3 */`, décrivez plutôt ce que fait l'algorithme en question !

Vous êtes encouragés à travailler à plusieurs, mais veillez à ***ne pas rendre du code copié*** sur vos camarades.

*
* *

Structures

Exercice 1.

Le fichier `structures.c` de l'archive du TP contient deux définitions de structures modélisant des étudiants et des équipes d'étudiants.

Pour les deux premières questions, vous pouvez manipuler des structures ou des pointeurs de structures au choix, et réserver la mémoire dans le tas ou dans la pile, du moment que vous libérez bien toute la mémoire allouée (n'oubliez pas **valgrind**).

Question 1. Rajoutez une fonction main, dans laquelle vous allez créer une équipe de 3 personnes :

- Camille, 23 ans
- Leila, 20 ans (la capitaine)
- Thibault, 22 ans

Question 2. Écrire deux fonctions d'affichage `void print_etu(etu_t* e);` `void print_equipe(equipe_t* e)`.

A partir de maintenant, nous allons manipuler exclusivement des pointeurs de structure, et on allouera toujours la mémoire dans le tas lorsque l'on crée une structure.

Question 3. Copiez le contenu du main (pour que je puisse voir ce que vous avez écrit à la première question), commentez la première copie, et modifiez l'autre pour n'utiliser que des pointeurs de structures et des malloc.

Question 4. Implémentez deux fonctions `void free_etu(etu_t* e);` `void free_equipe(equipe_t* e)` permettant de libérer la mémoire allouée, et rajoutez au main de quoi libérer la place allouée. N'oubliez pas de vérifier les fuites mémoires avec valgrind.

Nous allons maintenant implémenter une fonction `void agrandir_equipe(equipe_t* dst, equipe_t* src)` qui rajoute les membres de l'équipe source à l'équipe destination. Le capitaine de la nouvelle équipe sera le capitaine de la plus grande des deux équipes initiales.

Cette description de fonction est vague, et en particulier n'explique pas ce qui se passe au niveau de la mémoire : duplique-t-elle les structures des membres de l'équipe source, ou bien les deux équipes vont-elle partager des pointeurs communs vers les membres rajoutés ? La structure source est-elle encore valide après l'opération ? Ainsi de suite...

Question 5. Lisez le code suivant : quelles erreurs peuvent survenir selon les différents choix d'implémentation de la fonction `agrandir_equipe` ?

```
1  equipe_t* e1;
2  equipe_t e2;
3  /*
4    generation de e1 et e2
5  */
6
7  agrandir_equipe(e1, e2);
8
9  print_equipe(e1);
10 free_equipe(e1);
11
12 print_equipe(e2);
13 free_equipe(e2);
```

Question 6. Implémentez cette fonction selon la spécification plus précise suivante : la structure source devient invalide, elle est désallouée, et ses membres sont transférés à l'équipe destination.

Exercice 2.

On étudie dans cette exercice les structures en **listes chaînées**. Cette manière de concevoir des structures est très fréquente en informatique, et est à la base de nombreuses structures plus complexes que l'on verra plus tard en cours.

Une liste chaînée est constituée de maillons. Chaque maillon connaît son voisin de droite, ce qui permet de parcourir la liste de gauche à droite : on commence au maillon le plus à gauche, et on saute de voisin en voisin jusqu'à atteindre un maillon sans voisin de droite :

Il y a plusieurs manières d'implémenter des listes chaînées, on en propose une avec deux structures : une pour les maillons, et une pour la liste elle-même :

```c
struct maillon {
  struct maillon* suivant;
  /*
    attributs de données du maillon
  */
};
typedef struct maillon maillon_t;

struct liste {
  unsigned int taille;
  maillon_t* tete; // premier maillon
};
typedef struct liste liste_t;
```

Question 1. Quelle valeur peut-on donner à l'attribut `.suivant` d'un maillon en queue d'une liste ?

Question 2. Implémentez une structure de liste chaînée dont les maillons stockent des entiers. Implémentez une fonction `range` qui prend en entrée un entier n et renvoie une liste chaînée dont les maillons contiennent $0, 1, \ldots, n-1$ dans cet ordre. Essayez de le faire sans fonction récursive !

Question 3. Implémentez une fonction d'affichage listant le contenu de chaque maillon dans l'ordre, ainsi qu'une fonction de libération de mémoire. A nouveau, essayez d'implémenter ces fonctions avec des boucles uniquement, et sans récursivité.

Question 4. Écrivez une fonction `bool recherche(int x, liste_t * L)` qui détermine si la liste L contient au moins une fois l'élément x.

Question 5. Écrivez une fonction qui ajoute un élément à la fin d'une liste chaînée, et une autre qui ajoute un élément au début. Faîtes des dessins pour voir comment les différents pointeurs sont modifiés/ajoutés. En particulier, faîtes attention à ce qui peut arriver à l'attribut `tete` de la liste dans les deux cas.

Question 6. (Optionnel) Écrivez une fonction qui prend en entrée une liste chaînée et un entier, et retire le premier maillon contenant cet entier. A nouveau, faîtes des dessins et faîtes attention à bien mettre à jour la tête de liste.

Exercice 3.

L'archive du TP contient un fichier `promo_2013.txt` qui contient les noms des élèves qui étaient à votre place il y a 10 ans, avec le métier que chacun et chacune exerce à présent. Le fichier est formaté comme suit :

- Sur la première ligne, un entier n indiquant le nombre d'élèves
- Puis, n blocs de 3 lignes, chaque bloc correspondant à un élève, avec :
  - Sur une ligne, le prénom suivi du nom de famille (maximum 35 caractères)
  - Sur la ligne suivante, la date de naissance sous le format `JJ MM AAAA`
  - Sur la dernière ligne, le métier de cet élève aujourd'hui (maximum 35 caractères).

On propose de stocker les informations des élèves dans la structure suivante :

```c
struct eleve{
  char nom_complet[36];
  int jour, mois, annee; // date de naissance
  char metier[36];
};

typedef struct eleve eleve_t;
```

Nous allons commencer par écrire une fonction `eleve_t* lire_eleve (FILE* f)` qui lit les informations d'un/e élève dans le fichier f, et renvoie une structure contenant ces informations. Cette fonction renverra NULL si la fin du fichier est atteint.

Attention : lorsque vous scannez un entier avec le format `"%d"`, s'il y a un retour à la ligne après l'entier lu, il n'est pas consommé. Par exemple :

```c
fscanf(f, "%d", &n);
len = getline(ligne, &taille, f);
```

Si l'on exécute ce code et que le fichier `f` contient :

```
5
bonjour
```

alors la ligne scannée par `getline` sera vide, car après le `fscanf`, la tête de lecture du fichier se situe juste après le 5, avant le retour à la ligne.

Pour corriger cela, il faut consommer à la main le retour à la ligne. On peut donc utiliser un `char` comme poubelle :

```c
fscanf(f, "%d", &n);
char poubelle;
fscanf(f, "%c", &poubelle); // consomme le '\n'
len = getline(ligne, &taille, f);
```

Pour cet exercice, cette méthode fonctionnera bien car le fichier fourni est sous le bon format, et que ses retours à la ligne sont bien encodés par le caractère `'\n'`. Cependant, sur certains systèmes, notamment Windows, les sauts de ligne ne se font pas avec le caractère `'\n'` mais avec DEUX caractères : `'\r'` et `'\n'`. Le premier s'appelle le retour chariot, un terme qui vient des machines à écrire et qui désigne le fait de remettre le curseur tout à gauche du terminal (ou du papier pour les machines à écrire). Pour un fichier créé sous Windows, il faudrait donc faire :

```c
fscanf(f, "%d", &n);
char poubelle;
fscanf(f, "%c", &poubelle); // consomme le '\n'
fscanf(f, "%c", &poubelle); // consomme le '\r'
len = getline(ligne, &taille, f);
```

A nouveau, ce n'est pas nécessaire sur cet exercice car le fichier fourni a été créé sous Linux.

Question 1. En prenant en compte les informations précédentes, implémenter la fonction `eleve_t* lire_eleve (FILE* f)` permettant de lire dans un fichier les informations d'un/e élève.

Question 2. Écrire une fonction `eleve_t** lire_promo(char* filename, int* n)` qui renvoie un **tableau de pointeurs de structures** correspondant à la liste des élèves stockés dans le fichier de nom `filename`. Cette fonction stockera également dans `*n` le nombre d'élèves.

Question 3. Écrivez un programme qui permet d'afficher d'une manière un peu jolie les informations des élèves du fichier `promo_2013.txt`. N'oubliez pas de bien libérer **TOUTE** la mémoire allouée !

Livre dont *vous* êtes le héros ou la héroïne

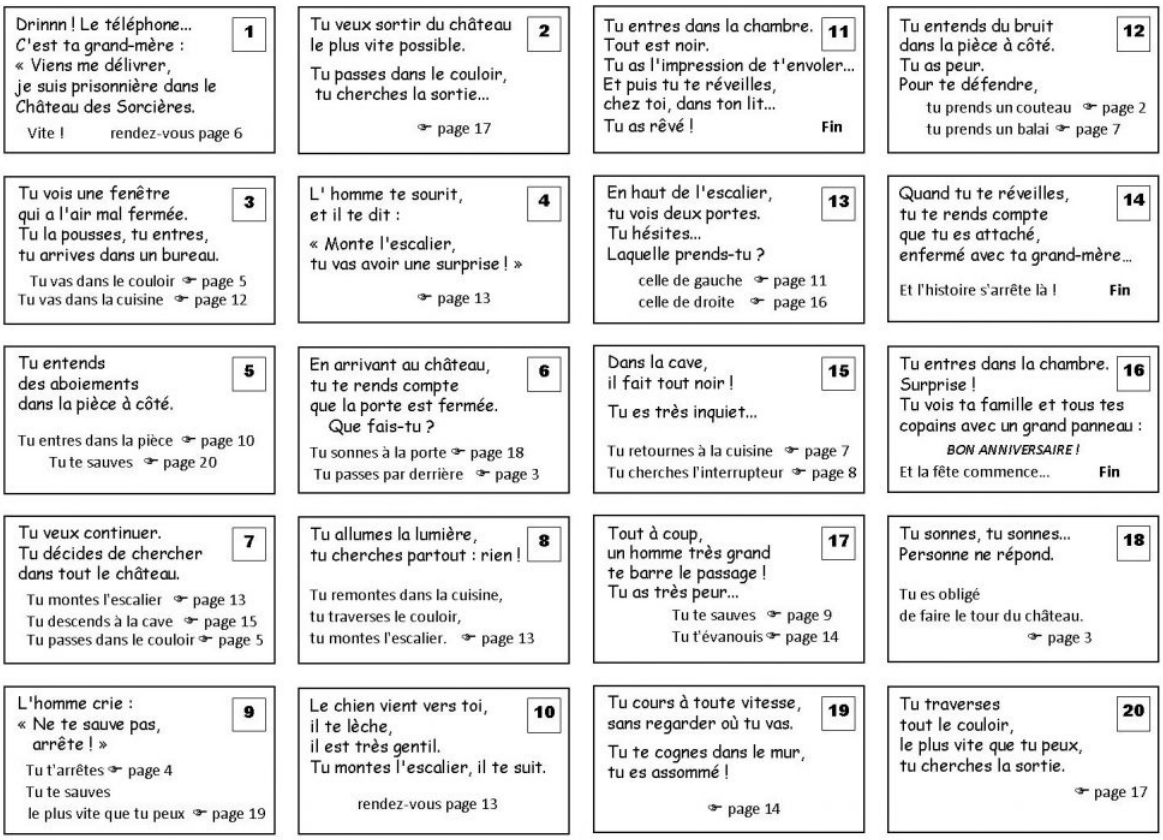

FIGURE 1 – Un LDVELH assez court

Exercice 4. Optionnel

Un ***Livre dont vous êtes le héro / la héroïne***, où LDVELH, est un type de livre interactif A chaque page, on est amené à faire un choix, et selon l'option choisie, à se rendre à une page particulière. Ainsi, un LDVELH ne se lit pas dans l'ordre mais en sautant de page en page en suivant les choix faits, en commençant à la page 1.

Le but de l'exercice est d'écrire un programme qui peut lire un LDVELH écrit dans un fichier texte sous un format bien spécifique, et en faire une version interactive, où l'utilisateur peut rentrer ses choix au clavier.

Pour représenter un LDVELH, on propose la modélisation suivante :

- Un **livre** est constitué de plusieurs pages, indexées par $0, 1, \ldots N-1$, où N est le nombre de pages
- Une **page** est constituée d'un texte, ainsi que de plusieurs choix, indexés par $0, 1, \ldots K-1$, où K est le nombre de choix
- Un **choix** est constitué d'un texte ainsi que du numéro de la page suivante lorsque l'on suit ce choix.

Pour représenter sous format texte un LDVELH, on écrit dans un fichier texte :

- Sur la première ligne, un entier n indiquant le nombre de pages total du livre

- Pour chaque page, on écrit :
  - Sur une ligne, le nombre de choix
  - Le texte de cette page sur une ligne
  - Pour chaque choix de la page, on écrit :
    - Le texte du choix sur une ligne
    - Le numéro de la page suivante

Une page marque la fin de l'aventure si, et seulement si, elle a 0 choix ; il peut y avoir plusieurs fins. La page d'indice 0 est la première page du livre.

Par exemple, les premières lignes du fichier "mamie.txt" représentant le LDVELH Fig. 1 seraient les suivantes (les commentaires ne seraient pas présents dans le fichier) :

```
20 // nombre de pages
1 // début de la page 0: un seul choix
Drinnn ! Le téléphone... C'est ta grand mère: "[...]" // texte de la page
Vite !  // texte de l'unique choix
5     // le choix mène à la page 5
1 // début de la page 1: un seul choix
Tu veux sortir du chateau [...] tu cherches la sortie... // texte de la page
   // texte du choix: cette ligne est vide
16 // le choix mène à la page 16
2 // début de la page 2: deux choix
Tu vois une fenêtre ... tu arrives dans un bureau. // texte de la page
Tu vas dans le couloir // texte du choix 1
4  // choix 1 -> aller page 4
Tu vas dans la cuisine // texte du choix 2
11 // choix 2 -> aller page 11
```

En C, on propose les structures suivantes pour représenter ces informations :

```c
struct CHOIX{
  char* texte;
  int page_suivante;
}

struct PAGE{
  char* texte;
  int n_choix;
  struct CHOIX** choix; //tableau de pointeurs vers des choix
}

struct LIVRE{
  int n_pages;
  struct PAGE** pages //tableau de pointeurs vers des pages
}
```

Vous pouvez utiliser typedef pour renommer ces structs si vous le souhaitez.

Question 1. Écrivez un fichier text.txt avec un exemple minimal de LDVELH (au moins 3 pages, au moins une page avec plusieurs choix)

Question 2. Écrivez une fonction `struct CHOIX* generer_choix(FILE* f)` qui, étant donné un fichier f, dont on suppose que la tête de lecture est positionnée à un endroit contenant un choix au format décrit plus haut, renvoie un pointeur de structure de type `struct CHOIX` en utilisant les données lues dans le fichier.

Question 3. Écrivez selon le même principe deux fonctions permettant de lire une page et un livre depuis un fichier :

```
1 struct PAGE* generer\_page(FILE* f);
2 struct LIVRE* generer\_livre(FILE* f);
```

Question 4. Écrivez une fonction `void free_livre (struct LIVRE* livre)` qui, étant donné un pointeur vers une structure struct LIVRE, libère toute la mémoire allouée à ce livre, y compris pour le texte. Vous pouvez écrire des fonctions auxiliaires pour vous aider (par exemple pour libérer les structures intermédiaires).

Question 5. En utilisant les fonctions précédentes, écrivez un programme prenant comme premier argument un nom de fichier texte, dont on suppose qu'il contient un LDVELH sous format texte, et qui permet à l'utilisateur de jouer/lire le livre.

L'exécution du programme doit ressembler vaguement à :

```
guillaume@MSI:~/prof/code/cyoa$
guillaume@MSI:~/prof/code/cyoa$ ./cyoa livre.txt
Nous sommes vendredi matin. Demain, il y a DS d'informatique. La journée doit se dérouler sans accroc afin que vous
ayez le temps de réviser ce soir, de dormir, et donc que vous puissiez obtenir une bonne note au DS. Il est 10h, le
cours de physique de Mme Goutelard vient de se terminer. Que faites-vous ?

  1. Vous allez en cours d'informatique.

  2. Vous n'allez pas en cours d'informatique, il fait beau et vous préférez aller vous balader sur les quais de la
Garonne.

Votre choix: 2
Vous n'allez pas en cours d'informatique, et vous ratez des informations capitales pour réviser. Vous n'êtes pas bie
n préparé pour le DS. GAME OVER
```

Vous trouverez dans l'archive du TP un fichier "livre.txt" écrit sous le format décrit plus haut, avec lequel vous pouvez tester votre programme une fois qu'il fonctionne sur votre exemple minimal. Vous y trouverez également un livre `multiverse.txt` co-écrit par Corentin, un élève de la promo 2022, qui a accepté de le laisser pour les promos suivantes.

Enfin, vous trouverez sur perso.ens-lyon.fr/guillaume.rousseau/ldvelh/ une version graphique du logiciel que vous avez écrit, permettant de visualiser les LDVELH sous forme d'un **graphe** (voir chapitre 13).