SCIENCES SORBONNE UNIVERSITÉ

Votre numéro d'anonymat : | | | |

Programmation et structures de données en C– LU2IN018

Examen du 14 juin 2021
1 heure 30
Aucun document n'est autorisé.

Les calculatrices, baladeurs et autres appareils électroniques sont interdits. Les téléphones mobiles doivent être éteints et rangés dans les sacs. Le barème sur 53 points (12 questions) n'a qu'une valeur indicative.

Les appels à `malloc` seront supposés réussir. Il ne sera pas nécessaire de vérifier leur valeur de retour. De la même manière, l'ouverture d'un fichier sera supposée réussir. Il ne sera pas nécessaire de vérifier que c'est bien le cas.

Le mémento qui vous a été distribué est reproduit à la fin de l'énoncé.

L'ensemble des structures et prototypes de fonctions est également rappelé à la fin de l'énoncé, sur une page détachable.

Locations de vacances

Les vacances approchent... Vous allez écrire des fonctions pour gérer les réservations d'un ensemble de logements et aider des clients à choisir des logements qui correspondent à leurs critères.

Les logements seront décrits par un nom, une adresse et un nombre de personnes. Ils contiendront également un ensemble d'équipements représentés sous la forme d'une liste chaînée de chaînes de caractères. Un logement contiendra aussi l'ensemble des réservations le concernant sous la forme d'une liste chaînée d'une structure contenant deux dates, les dates de début et de fin de la réservation. Les logements sont également stockés dans une liste chaînée. Cela donne les structures suivantes :

```
typedef struct _Equipement {
  char *nom;
  struct _Equipement *suivant;
} Equipement;

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

```
typedef struct _Reservation {
  Date debut;
  Date fin;
  struct _Reservation *suivant;
} Reservation;

typedef struct _Logement {
  char *nom;
  char *adresse;
  int nb_personnes;
  Equipement *equipement;
  Reservation *reservation;
} Logement;

typedef struct _ListeLogement {
  Logement *logement;
  struct _ListeLogement *suivant;
} ListeLogement;
```

Question 1 (6 points)

Écrivez une fonction de création d'un logement. Cette fonction transmet le nom et l'adresse, qui doivent être copiés. La liste d'équipements n'a pas besoin d'être dupliquée. Le logement nouvellement créé ne contiendra pas de réservation. Prototype :

```
Logement *creer_logement(char *nom, char *adresse, int nb_personnes,
   Equipement *equipement);
```

Réponse :

Question 2 (3 points)

Écrivez une fonction permettant de libérer les équipements. Elle devra libérer toute la mémoire associée à ces données, donc les chaînes de caractères et les éléments de la liste. Prototype :

```
void liberer_equipements(Equipement *leqpt);
```

La fonction prend en argument un pointeur vers le début de la liste d'équipements à libérer.

Réponse :

Question 3 (6 points)

Écrivez une fonction permettant de calculer le nombre de jours réservés pour un logement particulier. La fonction devra parcourir et ajouter les nombres de jours de chacune des réservations. Vous écrirez une fonction qui parcourra les réservations d'un logement et une qui calculera la durée d'une réservation particulière. Prototypes :

```
int nb_jours_reservation(Reservation *reservation);

int nb_jours_logement_reserve(Logement *logement);
```

La fonction `nb_jours_reservation` renvoie le nombre de jours réservés pour une réservation particulière. La fonction `nb_jours_logement_reserve` renvoie le nombre de jours réservés pour un logement particulier (elle doit donc parcourir toutes ses réservations en faisant appel à `nb_jours_reservation` pour chacune d'entre elles).

Le calcul de la durée d'une réservation pourra s'appuyer sur la fonction d'ajout d'un jour à une date et sur la fonction de comparaison de dates. Partant de la date de début, vous pourrez ajouter un jour jusqu'à dépasser la date de fin de la réservation et renvoyer le nombre de jours ainsi ajoutés. Une réservation ayant la même date en date de début et date de fin sera comptée pour 0 jour.

Prototypes (VOUS N'AVEZ PAS À ÉCRIRE CES FONCTIONS) :

```
int cmp_date(Date d1, Date d2);

Date ajouter_un_jour(Date d);
```

La fonction `cmp_date` renvoie -1 si `d1` est avant `d2`, 1 si `debut` est après `fin` et 0 si les dates sont les mêmes. La fonction `ajouter_un_jour` renvoie la date `d` à laquelle 1 jour a été ajouté.

Réponse :

Question 4 (6 points)

Écrivez une fonction permettant de vérifier si un logement est disponible sur une période donnée. Cette fonction procèdera de façon récursive. Prototype :

```
int est_disponible(Reservation *res, Date debut, Date fin);
```

Les réservations sont supposées classées par ordre croissant de date.

Si la liste des réservations passée en argument est vide, elle renverra 1, sinon, elle devra gérer les différents cas de figure :

— si la réservation `res` est avant la période `debut-fin` indiquée, on fait appel à la fonction en transmettant le suivant de `res` ;
— si la réservation commence avant la période et fini après, la fonction renvoie 0 ;
— si la réservation commence pendant la période, la fonction renvoie 0 ;
— si la réservation commence après la période, la fonction renvoie 1.

Réponse :

Question 5 (6 points)

Écrivez une fonction permettant d'insérer une réservation à sa place dans la liste des réservations (par ordre chronologique croissant). Prototype :

```
Reservation *inserer_reservation(Logement *logement, Date debut, Date
    fin);
```

Si le logement est disponible sur la période indiquée par `debut-fin`, une réservation sera créée et insérée en place dans la liste. La réservation créée sera renvoyée par la fonction, ou `NULL` si le logement n'est pas disponible à cette période.

Réponse :

Réponse :

Question 6 (3 points)

Écrivez une fonction permettant d'écrire une liste de réservations dans un fichier déjà ouvert en écriture. Prototype :

```
void ecrire_reservations(FILE *f,Reservation *reservation);
```

Exemple d'affichage :

```
30/12/2020 -> 3/1/2021
13/2/2021 -> 15/2/2021
```

Réponse :

Question 7 (3 points)

Écrivez une fonction permettant d'écrire une liste de logements dans un fichier dont le nom est transmis à la fonction. Prototype :

```
void ecrire_logements(char *nom_fichier, ListeLogement *llogement);
```

Cette fonction ouvrira le fichier en écriture puis fera appel à la fonction récursive suivante qui écrira les logements un à un (fonction que vous devrez aussi écrire). Prototype :

```
void ecrire_logements_rec(FILE *f, ListeLogement *llogement);
```

Le format est le suivant :

```
nom du logement
adresse du logement
nombre de personnes
<<<
équipement 1
équipement 2
...
>>>
Reservations:
reservation 1
reservation 2
...
===
```

Exemples :

```
Le nid douillet
Deauville
4
<<<
four
wifi
micro-onde
lave-vaisselle
>>>
Reservations:
30/12/2020 -> 03/01/2021
13/02/2021 -> 15/02/2021
===
Le coin tranquille
La Rochelle
6
<<<
TV
wifi
>>>
Reservations:
24/03/2021 -> 31/03/2021
===
```

Réponse :

Question 8 (6 points)

La lecture des logements va s'appuyer sur plusieurs fonctions pour lire les différentes informations d'un logement. Comme pour l'écriture, on définit une fonction permettant d'ouvrir le fichier en lecture et une fonction de lecture récursive. La fonction récursive de lecture (incomplète) est donnée ci-dessous :

```
ListeLogement *lire_logements_rec(FILE *f) {
  char nom[256];
  char adresse[256];
  int nb_pers;
  Equipement *eqpt=NULL;

  if (fgets(nom, 256, f)==NULL) {
    return NULL;
  }
  nom[strlen(nom)-1]='\0';
  fgets(adresse,256, f);
```

```
  adresse[strlen(adresse)-1]='\0';
  fscanf(f,"␣%d\n", &nb_pers);

  eqpt=lire_equipements(f);

  Logement *l=creer_logement(nom, adresse, nb_pers, eqpt);

  // lecture des réservations

  /* A COMPLETER : appel à la fonction de lecture des réservation */


  ListeLogement *llogement=(ListeLogement *)malloc(sizeof(
     ListeLogement));
  llogement->logement=l;

  llogement->suivant=lire_logements_rec(f);
  return llogement;
}
```

Indiquez comment appeler la fonction de lecture des réservations dans la fonction récursive de lecture des logements (contenu de la ligne `A COMPLETER`) et écrivez cette fonction. Prototype :

```
void lire_reservations(FILE *f, Logement *l);
```

Cette fonction de lecture va lire toute la partie liée aux réservations (à partir du début de la ligne contenant `Reservations:`) et les insérer dans la liste correspondante à l'aide de la fonction précédemment écrite. Le nombre de réservations est quelconque et n'est pas connu à l'avance. La lecture doit être poursuivie jusqu'à rencontrer la ligne contenant ===.

Réponse :

Réponse :

Question 9 (6 points)

Pour aider les vacanciers à choisir un logement, nous allons utiliser un arbre de décision. Pour rappel, un arbre de décision est un arbre binaire. À chaque noeud interne de l'arbre, une décision est associée. La prise de décision consiste à partir de la racine, poser la question associée puis poursuivre, de façon récursive, avec le sous-arbre correspondant à la réponse (oui/non), jusqu'aux feuilles de l'arbre qui contiennent le résultat de la décision. Dans le cas présent, la décision portera sur la présence ou non d'un équipement. Les feuilles contiendront la liste des logements vérifiant toutes les réponses qui ont permis de les atteindre (présence ou absence des équipements rencontrés pendant le parcours de l'arbre).

Structure de données :

```
typedef struct _ArbreDecision {
  char *equipement;
  ListeLogement *logements;
  struct _ArbreDecision *present;
  struct _ArbreDecision *absent;
} ArbreDecision;
```

Écrivez la fonction qui va poser des questions à l'utilisateur en fonction d'un arbre donné et qui va, lorsqu'une feuille est atteinte, afficher le résultat (le nom et l'adresse de chaque logement sont affichés sur une même ligne). Si le noeud en cours n'est pas une feuille, la fonction demande à l'utilisateur s'il souhaite disposer de l'équipement indiqué. Si l'utilisateur répond 'o', la recherche continue avec le sous-arbre `present`, sinon, elle continue avec le sous-arbre `absent`.

Exemple d'interaction :

```
Voulez-vous un logement avec "wifi" ? (o/n) n
Voulez-vous un logement avec "lave-vaisselle" ? (o/n) o
Voulez-vous un logement avec "TV" ? (o/n) o
```

```
Logements correspondants à vos critères:
La cabane du pirate, Saint Malo
Le chalet blanc, Chamonix
```

Les deux logements affichés sont les seuls à avoir un lave-vaisselle, une TV et pas de wifi. L'arbre binaire de décision a une racine portant sur l'équipement "wifi", son sous-arbre `absent` a pour équipement "lave-vaisselle" qui a lui un sous-arbre `present` avec l'équipement "TV" qui a enfin un sous-arbre `present` qui est une feuille (ses propres sous-arbres `absent` et `present` valent `NULL`) et contient une liste de logements avec les deux logements affichés.

Prototype :

```
void trouver_logement(ArbreDecision *ad);
```

Réponse :

Question 10 (3 points)

Écrivez la fonction de libération de la mémoire associée à un arbre. Cette fonction devra libérer toute la mémoire associée à un arbre à l'exception des logements (les listes de logements devront être libérées, mais pas les logements qu'elles contiennent).

Prototype :

```
void liberer_arbre(ArbreDecision *ad);
```

Réponse :

Question 11 (3 points)

Écrivez une fonction main qui va lire les logements contenus dans le fichier "mes_logements.txt", créer un arbre de décision à partir de cette liste de logements puis chercher le logement correspondant aux critères d'un vacancier avant de libérer la mémoire allouée. Vous pourrez pour cela vous appuyer sur les fonctions précédemment définies, ainsi que sur les fonctions de création d'un arbre et de libération de la mémoire (voir annexe).

Réponse :

Question 12 (2 points)

QUESTION BONUS

Les équipements, les réservations et les logements sont stockés dans des listes chaînées. Il serait donc intéressant d'utiliser une bibliothèque générique de listes chaînées plutôt que de répéter le code pour l'insertion, l'affichage, l'écriture et la libération de ces ensembles de données. Expliquez, sans écrire de code, comment se passerait la libération des données : quel est le principe de la libération générique ? Est-ce que le traitement de toutes les données (équipements, réservations et logements) serait le même ? Si non, préciser ce qu'il faudrait faire dans chaque cas.

Réponse :

Mémento de l'UE LU2IN018

Ce document a pour vocation de présenter un bref descriptif des fonctions offertes par les bibliothèques standards et qui sont susceptibles d'être utilisées dans l'UE.

Entrées - sorties

Prototypes disponibles dans `stdio.h`.

Entrées, sorties formatées

```
int printf(const char *format, ...);
```

La fonction `printf` écrit sur la sortie standard (par défaut le terminal), un message correspondant au texte spécifié dans la chaîne `format`. Le texte affiché peut contenir des portions *variables* spécifiées par des codes de conversion précédés par le caractère `%`. Les arguments suivant `format` doivent correspondre (ordre et type) aux codes de conversion présents. Voici quelques codes de conversion courants :

- `%d` : entier
- `%c` : caractère
- `%s` : chaîne de caractères
- `%f` : nombre réel

`printf` renvoie le nombre de caractères imprimés et la chaîne de format peut contenir des codes de contrôle permettant le formatage comme `\n` pour forcer le passage à la ligne et `\t` pour insérer une tabulation.

```
int scanf (const char *format, ...);
int sscanf(const char *entree, const char *format, ...);
```

Les fonctions `scanf` et `sscanf` permettent de saisir et analyser un texte saisi sur l'entrée standard, par défaut le clavier (`scanf`) ou depuis une chaîne de caractères passée en argument (`sscanf`). Le texte devra respecter le `format` spécifié et les arguments suivants doivent correspondre à des pointeurs sur des variables de type appropriés. Les codes de conversion son identiques à ceux de `printf`.

Entrées, sorties caractères

```
int getchar(void);
```

Lit un caractère dans le flux d'entrée standard (par défaut le clavier). La fonction retourne le code **EOF** en cas d'erreur, le caractère lu dans le cas contraire.

```
int putchar(int c);
```

Affiche un caractère dans le flux de sortie standard (par défaut le terminal). La fonction retourne le code **EOF** en cas d'erreur, le caractère lu dans le cas contraire.

```
int puts(const char *s);
```

Affiche une chaîne de caractères dans le terminal et passe à la ligne, renvoie **EOF** en cas d'erreur.

Fichiers

Prototypes disponibles dans `stdio.h`.

```
FILE *fopen(const char *path, const char *mode);
```

Ouvre un fichier dont le chemin est spécifié par la chaîne `path` et retourne un pointeur de type `FILE *` (`NULL` en cas d'échec). L'argument `mode` permet de spécifier le type d'accès à réaliser sur le fichier :

- `[r]` pour un accès en lecture,
- `[w]` pour un accès en écriture et le contenu précédent du fichier est écrasé,
- `[a]` pour un accès en écriture, le contenu du fichier est préservé et les écritures sont effectuées à la suite des contenus déjà présents.

En cas d'erreur la fonction retourne le code **NULL** sinon un pointeur vers le fichier ouvert.

```
int fclose(FILE *fp);
```

Cette fonction provoque la fermeture du fichier pointé par `fp`. En cas d'erreur la fonction retourne le code **EOF** sinon `0`.

```
int fprintf(FILE *stream, const char *format, ...);
```

Identique à `printf` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fscanf(FILE *stream, const char *format, ...);
```

Identique à `scanf` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
int fputc(int c, FILE *stream);
```

Identique à `putchar` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fputs(const char *s, FILE *stream);
```

Identique à `puts` mais l'argument `stream` permet de spécifier le flux de sortie et il n'y a pas d'ajout de passage à la ligne.

```
int getc(FILE *stream);
```

Identique à `getchar` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
char *fgets(char *s, int size, FILE *stream);
```

Lit au plus `size-1` octets dans le flux `stream`. La lecture s'arrête dès qu'un passage à la ligne est rencontré. Les octets lus sont stockés dans `s`. La fonction retourne `s` en cas de succès et `NULL` en cas d'erreur.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Lecture binaire de `nmemb` éléments de `size` octets dans le fichier `stream`. Les données lues sont stockées en mémoire à partir de l'adresse `ptr`. La fonction retourne le nombre d'éléments effectivement lus.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

Écriture de `nmemb` éléments de `size` octets dans le fichier `stream`. Les données à écrire sont lues en mémoire à partir de l'adresse `ptr`. La fonction retourne le nombre d'éléments effectivement écrits.

Chaînes de caractères

Prototypes disponibles dans `string.h`.

Une chaîne de caractères correspond à un tableau de caractère et doit contenir un marqueur de fin `\0`.

```
size_t strlen (const char *s);
```

Renvoie la longueur d'une chaîne de caractères (marqueur de fin `\0` non compris).

```
int strcmp (const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

Comparaison entre chaînes de caractères éventuellement limité aux `n` premiers caractères. La valeur retournée est :

- 0 si les deux chaînes sont identiques,
- négative si `s1` précède `s2` dans l'ordre lexicographique (généralisation de l'ordre alphabétique),
- positive sinon.

```
char *strcpy (char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

Copie le contenu de la chaîne `src` dans la chaîne `dest` (marqueur de fin `\0` compris). La chaîne `dest` doit avoir précédemment été allouée. La copie peut être limitée à `n` caractères et la valeur retournée correspond au pointeur de destination `dest`.

```
void *memcpy(void *dest, const void *src, size_t n);
```

Copie `n` octets à partir de l'adresse contenue dans le pointeur `src` vers l'adresse stockée dans `dest`. `dest` doit pointer vers une zone mémoire préalablement allouée et de taille suffisante. `memcpy` renvoie la valeur de `dest`.

```
size_t strlen(const char *s);
```

Retourne le nombre de caractères de la chaîne `s` (marqueur de fin `\0` non compris).

```
char *strdup(const char *s);
```

Cette fonction permet de dupliquer une chaîne de caractères, elle retourne un pointeur vers la chaîne nouvellement allouée. La nouvelle chaîne pourra être libérée avec la fonction `free`.

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Ajoute la chaîne `src` à la suite de la chaîne `dst`. La chaîne `dest` devra avoir été allouée et être de taille suffisante. La fonction retourne `dest`.

```
char *strstr(const char *haystack, const char *needle);
```

La fonction renvoie un pointeur sur la première occurrence de la sous-chaîne `needle` rencontrée dans la chaîne `haystack`. Si la chaîne recherchée n'est pas présente, la fonction retourne `NULL`.

Conversion de chaînes de caractères

Prototypes disponibles dans `stdlib.h`.

```
int atoi(const char *nptr);
```

La fonction convertit le début de la chaîne pointée par `nptr` en un entier de type `int`.

```
double atof(const char *nptr);
```

Cette fonction convertit le début de la chaîne pointée par `nptr` en un `double`.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Convertit le début de la chaîne `nptr` en un entier long. l'interprétation tient compte de la `base` et la variable pointée par `endptr` est affectée avec l'adresse du premier caractère invalide (au sens de la conversion).

Allocation dynamique de mémoire

Prototypes disponibles dans `stdlib.h`.

```
void *malloc(size_t size);
```

Alloue `size` octets de mémoire et retourne un pointeur générique correspondant à l'adresse du premier octet de la zone, renvoie `NULL` en cas d'échec.

```
void *realloc(void *ptr, size_t size);
```

Permet de modifier la taille d'une zone de mémoire allouée dynamiquement. `ptr` doit correspondre à l'adresse du premier octet de la zone précédemment allouée par `malloc` ou `realloc`. `size` correspond à la taille en octet de la nouvelle zone allouée. `realloc` garantie que la nouvelle zone contiendra les données présentes dans la zone initiale.

```
void free(void *ptr);
```

Libère une zone mémoire allouée dynamiquement. `ptr` doit correspondre à l'adresse du premier octet de la zone précédemment allouée par `malloc` ou `realloc`.

La liste des fonctions du programme considéré est indiquée ci-après (fichier "reservation.h"). Certaines fonctions ne sont pas à écrire. Ces fonctions peuvent tout de même être utilisées et considérées comme disponibles.

```
#ifndef _RESERVATION_H_
#define  _RESERVATION_H_

#include <stdio.h>

extern char nb_jours_mois[];

typedef struct _Equipement {
  char *nom;
  struct _Equipement *suivant;
} Equipement;

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

typedef struct _Reservation {
  Date debut;
  Date fin;
  struct _Reservation *suivant;
} Reservation;

typedef struct _Logement {
  char *nom;
  char *adresse;
  int nb_personnes;
  Equipement *equipement;
  Reservation *reservation;
} Logement;

typedef struct _ListeLogement {
  Logement *logement;
  struct _ListeLogement *suivant;
} ListeLogement;

typedef struct _ArbreDecision {
  char *equipement;
  ListeLogement *logements;
  struct _ArbreDecision *present;
  struct _ArbreDecision *absent;
} ArbreDecision;


/* Creer un logement (Q1) */
Logement *creer_logement(char *nom, char *adresse, int
    nb_personnes, Equipement *equipement);

/* Creer une reservation */
Reservation *creer_reservation(Date debut, Date fin);

/* Ecrire/afficher les reservations (Q2) */
void ecrire_reservations(FILE *f,Reservation *reservation
    );

/* Comparaison de deux dates */
int cmp_date(Date d1, Date d2);

/* Ajouter un jour a une date */
Date ajouter_un_jour(Date d);

/* Calculer le nombre de jours d'une reservation */
int nb_jours_reservation(Reservation *reservation);

/* Nombre de jours de reservation d'un logement (Q3) */
int nb_jours_logement_reserve(Logement *logement);

/* Verifier si un logement est disponible (Q4) */
int est_disponible(Reservation *res, Date debut, Date fin
    );

/* Inserer une reservation (Q5) */
Reservation *inserer_reservation(Logement *logement, Date
     debut, Date fin);

/* Ecrire les logements, fonction recurrente (Q6) */
void ecrire_logements_rec(FILE *f, ListeLogement *
    llogement);

/* Ecrire les logements (Q6) */
```

```
void ecrire_logements(char *nom_fichier, ListeLogement *
    llogement);

/* Lecture des équipements (Q8) */
Equipement *lire_equipements(FILE *f);

/* Lecture des réservations */
void lire_reservations(FILE *f, Logement *l);

/* Lire les logements, version recurrente */
ListeLogement *lire_logements_rec(FILE *f);

/* Lire les logements (Q7) */
ListeLogement *lire_logements(char *nom_fichier);

/* Liberer les equipements (Q2) */
void liberer_equipements(Equipement *leqpt);

/* Liberer les reservations */
void liberer_reservations(Reservation *res);

/* Liberer les logements */
void liberer_logements(ListeLogement *l);

/* Creer un arbre de decision depuis une liste de
   logements */
ArbreDecision *creer_arbre(ListeLogement *llogement);

/* Afficher un arbre de decision */
void afficher_arbre(ArbreDecision *ad, int espace);

/* Trouver le ou les logements correspondants a des
   criteres donnes (Q9) */
void trouver_logement(ArbreDecision *ad);

/* Liberer un arbre */
void liberer_arbre(ArbreDecision *ad);

#endif
```