SCIENCES SORBONNE UNIVERSITÉ

Votre numéro d'anonymat : | | | |

Programmation et structures de données en C– 2I001

Examen du 16 janvier 2019
2 heures
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 63 points (10 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.

LOTO®

Le LOTO® est un jeu de hasard existant depuis 1976. Plusieurs millions de joueurs s'y adonnent régulièrement. Le principe est de choisir un ensemble de numéros entre 1 et 49. Des boules sur lesquelles sont inscrits des numéros sont tirées au sort plusieurs fois par semaine. Les joueurs ayant validé une grille de jeu pour un tirage, gagnent un montant dépendant du nombre de numéros choisis parmi les boules ayant été tirées. Les principaux gagnants sont ceux qui ont indiqué tous les chiffres tirés (ils peuvent gagner plusieurs millions d'euros). On parle de gagnants de rang 1. Les personnes à qui il manque un des numéros tirés sont appelés gagnants de rang 2 (leurs gains sont inférieurs), et ainsi de suite.

Nous allons considérer ici une version simplifiée de ce jeu dans laquelle seulement 5 numéros sont tirés et n'autorisant les joueurs à ne choisir que 5 numéros (la version officielle contient des numéros particuliers en supplément). Par ailleurs, toutes les combinaisons seront considérées dans l'ordre croissant (i.e. cocher les cases 3 puis 5 puis 3 puis 1 puis 4 donne la combinaison "1 2 3 4 5"). De même, pour faciliter les comparaisons, les tirages seront considérés comme ordonnés même si bien sûr les boules peuvent être tirées dans n'importe quel ordre.

Les fonctions que vous allez écrire dans la suite permettent de stocker un ensemble, potentiellement large, de données associées aux tickets des joueurs. Vous allez en particulier écrire les fonctions permettant de parcourir ces ensembles de données pour trouver les gagnants. Après quelques questions sur la structure de base pour stocker les tickets, vous utiliserez des listes "compressées", puis des arbres.

1 Gestion des tickets

Les tickets de jeu seront gérés au travers d'une structure de données dédiée. Cette structure contient les 5 entiers choisis par le joueur. Ces 5 numéros sont tous différents et classés par ordre croissant. Ils sont situés entre 1 et 49. Nous les représenterons donc avec des `char`, qui sont suffisants pour représenter de telles valeurs. Un ticket contient également un identifiant et le code postal de la maison de la presse ou du bureau de tabac où le billet a été validé. Pour pouvoir stocker les tickets dans une liste chaînée, un dernier champ contient un pointeur permettant de stocker, s'il y a lieu, l'adresse de l'élément suivant. Cela donne la structure suivante :

```
typedef struct _ticket {
  char numeros[5];
  int id;
  int code_postal;
  struct _ticket *suivant;
} Ticket;
```

Question 1 (6 points)

Écrivez une fonction permettant de calculer le nombre de numéros en commun entre deux tableaux de même taille et triés par ordre croissant. Prototype :

```
int nb_communs(const char num1[], const char num2[], int taille);
```

Solution:

```
int nb_communs(const char num1[], const char num2[], int taille) {
  unsigned int i, j;
  int res=0;
  for (i=0;i<taille;i++) {
    for (j=0;j<taille;j++) {
      if (num2[j]>num1[i]) break; // les tableaux sont tries
      if(num1[i]==num2[j]) {
        res++;
        break; //il faut passer au prochain i
      }
    }
  }
  return res;
}
```

Question 2 (3 points)

Écrivez une fonction permettant d'allouer la mémoire pour une structure de type `Ticket` et d'initialiser ses différents champs. Attention, tous les champs doivent être initialisés. Prototype :

```
Ticket *creer_ticket(const char numeros[5], int id, int code_postal);
```

Solution:

```
Ticket *creer_ticket(const char numeros[5], int id, int code_postal)
    {
  Ticket *t=(Ticket *)malloc(sizeof(Ticket));
```

```
  unsigned int i=0;
  for (i=0;i<5;i++)
    t->numeros[i]=numeros[i];
  t->id=id;
  t->code_postal=code_postal;
  t->suivant=NULL;
  return t;
}
```

Question 3 (6 points)

La fonction suivante doit permettre de concaténer deux listes de tickets. Vous ne ferez pas de copies des tickets et insèrerez la liste indiquée par `tickets2` à la fin de la liste indiquée par `ticket1`. Prototype :

```
Ticket *ajouter_tickets(Ticket *tickets1, Ticket *tickets2);
```

Solution:

```
Ticket *ajouter_tickets(Ticket *tickets1, Ticket *tickets2) {
  if (tickets1==NULL) return tickets2;
  Ticket *tmp=tickets1;
  while(tmp->suivant) {
    tmp=tmp->suivant;
  }
  tmp->suivant=tickets2;
  return tickets1;
}
```

Question 4 (3 points)

Écrivez la fonction permettant de libérer la mémoire associée à une liste de tickets. Prototype :

```
void liberer_tickets(Ticket *tickets);
```

Solution:

```
void liberer_tickets(Ticket *tickets) {
  Ticket *tmp;
  while(tickets) {
    tmp=tickets->suivant;
    free(tickets);
    tickets=tmp;
  }
}
```

Question 5 (6 points)

Les tickets sont stockés dans un fichier de la façon suivante :

```
<num1> <num2> <num3> <num4> <num5> <id> <code_postal>
```

Exemple :

```
2 8 16 25 37 10001 55184
15 18 23 29 44 10002 79241
9 14 23 29 35 10003 06138
1 2 7 18 45 10004 02467
```

```
16 22 38 47 48 10005 06830
7 16 38 39 46 10006 63257
2 32 34 44 46 10007 46357
5 7 14 31 42 10008 64508
```

Écrivez une fonction permettant de lire le contenu d'un fichier dont le nom est transmis en argument de la fonction. Cette fonction renverra la liste des tickets qui aura été lue. Prototype :

```
Ticket *lire_tickets(const char *fichier);
```

Pour rappel, le code du format pour lire un entier de type `char` est `hhd`. Vous prendrez soin de vérifier que le format du fichier est correct. Pour rappel, la fonction `scanf` renvoie le nombre d'éléments qui ont été effectivement lus. Si le format est incorrect, vous afficherez le message `Erreur de format` et vous quitterez le programme.

Solution:

```
Ticket *lire_tickets(const char *fichier) {
  FILE *f=fopen(fichier, "r");
  Ticket *res=NULL;
  char buffer[100];
  char numeros[5];
  int id;
  int code_postal;
  while(fgets(buffer, 100, f)) {
    if(sscanf(buffer, "%hhd␣%hhd␣%hhd␣%hhd␣%hhd␣%d␣%d",
              numeros, numeros+1, numeros+2, numeros+3, numeros+4,
              &id, &code_postal)!=7) {
      printf("Erreur␣de␣format\n");
      exit(1);
    }
    res=ajouter_tickets(res,creer_ticket(numeros,id,code_postal));
  }
  fclose(f);
  return res;
}
```

2 Liste de tickets

Dans les questions précédentes, les numéros sont stockés sous la forme d'une liste chaînée. Il arrive fréquemment que plusieurs personnes choisissent le même ensemble de numéros. Pour que la recherche des gagnants soit plus rapide, nous allons définir une autre structure de liste chaînée dans laquelle chaque ensemble de numéros n'apparait d'une seule fois. La donnée associée à un ensemble de numéros sera la liste chainée des tickets correspondant à ces numéros [1].

1. Remarque : la liste des numéros apparaitra également dans chacun des tickets de la donnée. Nous garderons tout de même la structure `Ticket` inchangée pour simplifier.

Cela donne la structure suivante :

```
typedef struct _liste_par_numero {
  char numeros[5];
  Ticket *ticket;
  struct _liste_par_numero *suivant;
} Liste_par_numero;
```

Question 6 (6 points)

Écrivez la fonction permettant de créer une liste par numéro à partir d'une liste de tickets. Prototype :

```
Liste_par_numero *creer_liste_par_numero(Ticket *tickets);
```

Cette fonction parcourra la liste des tickets et, pour chacun d'eux, vérifiera si les numéros correspondants sont déjà dans la liste par numéro. Vous pouvez utiliser pour cela la fonction de la question 1. Si c'est le cas, le ticket sera ajouté à la liste des tickets correspondants (vous pouvez choisir un ajout en début ou en fin de liste), sinon, le ticket sera ajouté à un nouvel élément de type `Liste_par_numero`. Vous pourrez utiliser pour cela la fonction de prototype suivant, qui crée un élément de ce type à partir d'un numéro et d'une liste de tickets (vous n'avez pas à l'écrire) :

```
Liste_par_numero *creer_element_lpn(const char numeros[5], Ticket *
   ticket);
```

Solution:

```
Liste_par_numero *creer_liste_par_numero(Ticket *tickets) {
  Liste_par_numero *res=NULL,*nres=NULL;
  Liste_par_numero *tmp=NULL;

  int ajouter_en_fin;
  while(tickets) {
    tmp=res;

    ajouter_en_fin=1;
    while (tmp) {
      if (nb_communs(tickets->numeros,tmp->numeros,5)==5) {
        tmp->ticket=ajouter_tickets(tmp->ticket,creer_ticket(tickets
           ->numeros, tickets->id, tickets->code_postal));
        ajouter_en_fin=0;
        break;
      }

      tmp=tmp->suivant;
    }
    if(ajouter_en_fin) {
      if(res==NULL) {
        // Ajout a la liste vide
        res=creer_element_lpn(tickets->numeros, tickets);
      }
      else {
        nres=creer_element_lpn(tickets->numeros, tickets);
        nres->suivant=res;
```

```
        res=nres;
      }
    }

    tickets=tickets->suivant;
  }
  return res;
}
```

Question 7 (6 points)

Écrivez maintenant la fonction permettant de déterminer les gagnants d'un tirage. La fonction prend en argument la liste des tickets joués sous forme de `Liste_par_numero` ainsi que les numéros tirés et le nombre de rangs souhaités. Prototype :

```
Ticket ** chercher_gagnants(Liste_par_numero *liste, const char
   numeros[], int nb_rangs);
```

La fonction renvoie un tableau de listes de ticket qui aura été alloué dans cette fonction. Ce tableau est de taille `nb_rangs`. Le contenu de la case 0 sera les gagnants du premier rang (ceux qui ont trouvé tous les numéros, autrement dit qui ont 0 mauvais numéros), le contenu de la case 1 est la liste des gagnants du rang 2 (ceux qui ont 1 erreur dans la liste), ... Vous prendrez soin de créer une copie des tickets avant de les ajouter à la liste correspondant à leur rang.

Solution:

```
Ticket ** chercher_gagnants(Liste_par_numero *liste, const char
   numeros[], int nb_rangs) {
  Ticket **res=(Ticket **)malloc(nb_rangs*sizeof(Ticket *));
  int i, nbc;
  for (i=0;i<nb_rangs;i++) {
    res[i]=NULL;
  }

  while (liste) {
    nbc=nb_communs(numeros, liste->numeros,5);
    int rang=5-nbc;
    if(rang<nb_rangs) {
      Ticket *lt=liste->ticket;
      while (lt) {
        res[rang] = ajouter_tickets(res[rang], creer_ticket(lt->
           numeros,lt->id,lt->code_postal));
        lt=lt->suivant;
      }
    }
    liste=liste->suivant;
  }

  return res;
}
```

3 Stockage des tickets dans un arbre

La fonction de recherche s'appuyant sur une liste s'avère trop lente par rapport aux attentes. Pour accélérer la recherche, les tickets vont être stockés sur un arbre. Structure :

```
typedef struct _arbre_par_numero {
  Ticket *tickets;
  struct _arbre_par_numero *fils[NB_NUM];
} Arbre_par_numero;
```

Chaque nœud de l'arbre aura 49 fils. Les ensembles de numéros seront stockés en créant un nœud par numéro et en le rangeant dans la case d'indice n-1 si n est le numéro correspondant (car le tableau commence à l'indice 0, et non au 1). L'arbre sera de hauteur 5 et le chemin dans l'arbre jusqu'à une feuille indiquera les 5 numéros choisis [2]. Les tickets correspondant à cette séquence seront stockés sur les feuilles de l'arbre (les nœuds internes auront une liste de tickets associée à `NULL`).

Exemple : pour l'ensemble de numéros `(3,9,16,32,44)`, le numéro 3 est indiqué par un nœud stocké dans la case 2 de la racine de l'arbre (3-1=2). Ce nœud a lui-même un fils dans la case 8 (9-1), qui a lui-même un fils dans la case 15 (16-1), qui a lui-même un fils dans la case 31 (32-1), qui a lui-même un fils dans la case 43 (44-1). Ce dernier est une feuille de l'arbre et contient donc la liste des tickets associés à ces 5 numéros.

Question 8 (9 points)

Écrivez les 2 fonctions permettant de créer un arbre à partir d'une liste de tickets. La fonction de prototype :

```
Arbre_par_numero *ajouter_tickets_arbre(Arbre_par_numero *arbre,
   Ticket *tickets);
```

parcourt la liste de tickets (liste non compressée de la partie 1) et les ajoute un à un à l'arbre avec la fonction suivante.

L'ajout d'un ticket dans l'arbre se fait avec une fonction *récursive*. Cette fonction ajoutera un numéro du ticket à chaque niveau de l'arbre. Attention, la récursion se fera donc à la fois sur l'arbre et sur l'indice dans le tableau des numéros : il faut descendre dans l'arbre et avancer dans le tableau des numéros. La fonction prend en argument l'arbre dans lequel ajouter le ticket, le ticket à ajouter ainsi que l'indice du numéro à ajouter (0 au départ et 5 à la fin : il n'y a plus de numéro à ajouter). Vous prendrez soin d'ajouter une copie du ticket à l'arbre. Prototype :

```
Arbre_par_numero *ajout_1_ticket_rec(Arbre_par_numero *arbre, Ticket
   *ticket, int indice_numero);
```

Solution:

```
Arbre_par_numero *ajout_1_ticket_rec(Arbre_par_numero *arbre, Ticket
    *ticket, int indice_numero) {
  if (indice_numero==5) {
    // fin de la recurrence
    if(arbre==NULL) {
      arbre=creer_arbre(ticket);
    }
```

2. Remarque : on cherche ici uniquement à optimiser le temps de recherche et l'arbre proposé est plus coûteux en mémoire que ce qui est réellement nécessaire (on pourrait se contenter de 48 cases sous la racine, 47 ensuite, etc.) mais cette optimisation alourdirait le code à écrire pour la recherche.

```
    else {
      arbre->tickets=ajouter_tickets(arbre->tickets,creer_ticket(
         ticket->numeros,ticket->id,ticket->code_postal));
    }
    return arbre;
  }

  if(arbre==NULL) {
    arbre=creer_arbre(NULL);
  }
  arbre->fils[(int)ticket->numeros[indice_numero]-1]=
     ajout_1_ticket_rec(arbre->fils[(int)ticket->numeros[
     indice_numero]-1], ticket, indice_numero+1);
  return arbre;
}
Arbre_par_numero *ajouter_tickets_arbre(Arbre_par_numero *arbre,
   Ticket *tickets) {
  while(tickets) {

    //    printf("Ajouter: %d %d %d %d %d\n",tickets->numeros[0],
       tickets->numeros[1],tickets->numeros[2],tickets->numeros[3],
       tickets->numeros[4]);
    arbre=ajout_1_ticket_rec(arbre,tickets,0);
    tickets=tickets->suivant;
  }
  return arbre;
}
```

Question 9 (9 points)

Écrivez maintenant les fonctions permettant de rechercher les gagnants au tirage. Comme précédemment, cette fonction prendra en argument l'arbre dans lequel les tickets sont rangés, les numéros tirés ainsi que le nombre de rangs souhaités. La fonction renvoie un tableau de listes de tickets qui aura été alloué dans la fonction. Prototype :

```
Ticket **chercher_gagnants_arbre(Arbre_par_numero *arbre, const char
   numeros[], int nb_rangs);
```

Cette fonction allouera le tableau de résultats, initialisera chacune de ses cases à `NULL` et fera appel à la fonction récurrente de recherche dans l'arbre de prototype suivant et que vous devez également écrire :

```
void chercher_arbre_rec(Arbre_par_numero *arbre, const char numeros
   [], int nb_rangs, Ticket **res, int indice_numero, int
   mauvais_numeros);
```

Cette fonction prend en argument l'arbre, les numéros tirés, le nombre de rangs souhaités, le tableau de résultats, le niveau auquel on se situe dans l'arbre (0 pour la racine) et le nombre de mauvais numéros parmi les numéros déjà pris en compte. Vous déterminerez vous même les cas de base. Dans le cas général, l'algorithme sera :

```
Pour chaque numéro i possible,
   Si le ieme fils n'est pas NULL:
        Si (i+1) existe dans le tableau numéros alors
             Faire l'appel récursif, avec un nombre de mauvais numéros \
inchangé (vous déterminerez vous même les autres arguments),
        Sinon,
             Faire l'appel récursif, avec un nombre de mauvais numéros \
augmenté de 1
```

Solution:

```
void chercher_arbre_rec(Arbre_par_numero *arbre, const char numeros
   [], int nb_rangs, Ticket **res, int indice_numero, int
   mauvais_numeros) {
  if (arbre==NULL) {
    //printf(" Arbre vide...\n");
    return;
  }
  if (mauvais_numeros>=nb_rangs) {
    //printf("Trop de mauvais numeros...\n");
    return;
  }
  if (indice_numero==5) {
    // La fin de l'arbre est atteinte
    Ticket *lt=arbre->tickets;
    //printf("Chercher_rec: trouve %d tickets pour le rang %d\n",
       compter_tickets(lt),mauvais_numeros+1);
    while (lt) {
      res[mauvais_numeros] = ajouter_tickets(res[mauvais_numeros],
         creer_ticket(lt->numeros,lt->id,lt->code_postal));
      lt=lt->suivant;
    }
  }
  else {
    int i,j,trouve;
    for (i=0;i<NB_NUM;i++) {
      trouve=0;
      for (j=0;j<5;j++) {
        if (i==numeros[j]-1) {
          trouve=1;
          break;
        }
      }
      if(trouve){
        //printf("Numero OK: i=%d ind=%d num[ind]=%d nbr=%d mauvais
           =%d \n",i,indice_numero,numeros[indice_numero], nb_rangs,
            mauvais_numeros);
        chercher_arbre_rec(arbre->fils[i], numeros, nb_rangs, res,
           indice_numero+1, mauvais_numeros);
```

```
      }
      else {
        //printf("Numero KO: i=%d ind=%d num[ind]=%d nbr=%d mauvais
           =%d\n",i,indice_numero,numeros[indice_numero], nb_rangs,
           mauvais_numeros);
        chercher_arbre_rec(arbre->fils[i], numeros, nb_rangs, res,
           indice_numero+1, mauvais_numeros+1);
      }
    }
  }
}

Ticket **chercher_gagnants_arbre(Arbre_par_numero *arbre, const char
    numeros[], int nb_rangs) {
  Ticket **res=(Ticket **)malloc(nb_rangs*sizeof(Ticket *));
  int i;
  for (i=0;i<nb_rangs;i++) {
    res[i]=NULL;
  }

  chercher_arbre_rec(arbre,numeros,nb_rangs,res,0,0);
  return res;

  }
```

Question 10 (9 points)

Écrire une fonction main. En s'appuyant sur les fonctions vues précédemment, cette fonction lira les tickets depuis un fichier nommé `tickets.txt`, créera l'arbre correspondant et cherchera les gagnants des 3 premiers rangs pour une combinaison arbitrairement donnée en dur dans le code du main. Les gagnants du 1er rang seront tous affichés. Pour les autres rangs, seul le nombre de gagnants sera affiché. Vous pourrez utiliser pour cela la fonction suivante (que vous n'avez pas besoin d'écrire) :

```
int compter_tickets(Ticket *tickets);
```

Vous prendrez soin de libérer toute la mémoire allouée. Pour libérer la mémoire associée à un arbre (incluant les tickets associés), vous pourrez utiliser la fonction suivante (que vous n'aurez pas à écrire non plus) :

```
void liberer_arbre(Arbre_par_numero *arbre);
```

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include "loto.h"

int main(void) {

  Ticket *tickets=lire_tickets("tickets.txt");
```

```
  char num[5]={12,28,29,36,40};
  int nbr=3;
  int i;
  Ticket **v;

  Arbre_par_numero *arbre=ajouter_tickets_arbre(NULL, tickets);

  v=chercher_gagnants_arbre(arbre,num,nbr);
  printf("Rang␣1:␣vainqueurs:\n");
  printf("Rang␣%d:␣%d␣vainqueurs\n",1,compter_tickets(v[0]));
  Ticket *tmp=v[0];
  while(tmp) {
    printf("%hhd␣%hhd␣%hhd␣%hhd␣%hhd␣%d␣%d\n",tmp->numeros[0],tmp->
       numeros[1],tmp->numeros[2],tmp->numeros[3],tmp->numeros[4],
       tmp->id, tmp->code_postal);
    tmp=tmp->suivant;
  }
  liberer_tickets(v[0]);
  for (i=1;i<nbr;i++) {
    printf("Rang␣%d:␣%d␣vainqueurs\n",i+1,compter_tickets(v[i]));

    liberer_tickets(v[i]);
  }
  free(v);

  liberer_arbre(arbre);
  liberer_tickets(tickets);
  return 0;
}
```

Mémento de l'UE 2I001

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` 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. Certaines fonctions ne sont pas à écrire. Ces fonctions peuvent tout de même être utilisées et considérées comme disponibles.

```
#ifndef _LOTO_H_
#define _LOTO_H_

#define NB_NUM 49

typedef struct _ticket {
  char numeros[5];
  int id;
  int code_postal;
  struct _ticket *suivant;
} Ticket;

// Q1: Nombre de valeurs identiques entre num1
   et num2
int nb_communs(const char num1[], const char
   num2[], int taille);

// Q2: creer un ticket
Ticket *creer_ticket(const char numeros[5], int
    id, int code_postal);

// Q3: ajouter une liste de tickets a une autre
    liste
Ticket *ajouter_tickets(Ticket *tickets1,
   Ticket *tickets2);

// Affichage des tickets
void afficher_tickets(Ticket *tickets);

// Compter le nombre de tickets dans une liste
int compter_tickets(Ticket *tickets);

// Q4: liberer la memoire associee a une liste
   de tickets
void liberer_tickets(Ticket *tickets);

// Q5: lire des tickets depuis un fichier et
   renvoyer la liste des tickets
Ticket *lire_tickets(const char *fichier);


typedef struct _liste_par_numero {
  char numeros[5];
  Ticket *ticket;
  struct _liste_par_numero *suivant;
} Liste_par_numero;

// Creer un element de liste classee par numero
Liste_par_numero *creer_element_lpn(const char
   numeros[5], Ticket *ticket);

// Calcul du nombre d'elements dans une liste
   par numero
int nb_elements(Liste_par_numero *ln);

// Q6: creer une liste par numero depuis une
   liste de tickets
Liste_par_numero *creer_liste_par_numero(Ticket
    *tickets);

// Liberer une liste par numero
void liberer_liste_par_numero(Liste_par_numero
   *liste);

// Q7: Chercher les gagnants a partir d'une
   liste classee par numeros
Ticket ** chercher_gagnants(Liste_par_numero *
```

```
   liste, const char numeros[], int nb_rangs);



typedef struct _arbre_par_numero {
  Ticket *tickets;
  struct _arbre_par_numero *fils[NB_NUM];
} Arbre_par_numero;

// Creer un noeud d'arbre
Arbre_par_numero *creer_arbre(Ticket *ticket);

// Liberer la memoire allouee a un arbre
void liberer_arbre(Arbre_par_numero *arbre);

// Calculer le nombre de noeuds
int nb_noeuds(Arbre_par_numero *arbre);

// Calculer le nombre de tickets dans un arbre
int nb_tickets(Arbre_par_numero *arbre);

// Afficher le contenu d'un arbre
void afficher_arbre(Arbre_par_numero *arbre,
   int niveau);

// Q8: Ajouter une liste de tickets a un arbre
Arbre_par_numero *ajout_1_ticket_rec(
   Arbre_par_numero *arbre, Ticket *ticket, int
    indice_numero);
Arbre_par_numero *ajouter_tickets_arbre(
   Arbre_par_numero *arbre, Ticket *tickets);

// Q9: Chercher les gagnants a partir d'un
   arbre
Ticket **chercher_gagnants_arbre(
   Arbre_par_numero *arbre, const char numeros
   [], int nb_rangs);
void chercher_arbre_rec(Arbre_par_numero *arbre
   , const char numeros[], int nb_rangs, Ticket
    **res, int indice_numero, int
   mauvais_numeros);
#endif
```