SCIENCES SORBONNE UNIVERSITÉ

Votre numéro d'anonymat : | | | |

Programmation et structures de données en C– LU2IN018

Examen du 18 décembre 2023
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 51 points (11 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.

Gestion de l'usure des paires de chaussures de course à pied

De nombreuses études scientifiques ont établi une corrélation entre l'usure des chaussures et le risque de blessures lors de la course à pied. Il est recommandé de changer ses chaussures de course à pied tous les 1000km environ. Léa pratique la course à pied depuis deux ans et elle possède plusieurs paires de chaussures dédiées à cette activité. Elle souhaiterait créer un outil permettant de stocker ses différentes sorties de course à pied, tout en surveillant l'usure de ses chaussures, afin de les remplacer avant tout risque de blessure.

1 Gestion des sorties effectuées avec une paire de chaussures de course a pied

Les sorties de course à pied de Léa sont représentées sous la forme de listes chaînées avec la structure suivante :

```
typedef struct _sortie {
    char* date;
    char *chaussure; /* texte décrivant la marque et le modèle de la
       paire de chaussure */
    float dist; /* nombre de kilomètres de la sortie en km */
    struct _sortie *suiv;
} Sortie;
```

Pour chaque sortie est enregistrée la date, représentée par une chaine de caractère au format anglais AAAA-MM-JJ. Ainsi la date du 03 Novembre 2023 est représentée par la chaine de caractère "2023-11-03", ce qui permettra ensuite de comparer les chaines de caractères et de classer les dates par ordre chronologique si besoin.

Associée à la sortie, sont aussi stockés, le nom de la paire de chaussures utilisée, la distance de la sortie, ainsi qu'un pointeur sur la sortie suivante.

Question 1 (3 points)

Écrivez la fonction permettant de créer une nouvelle sortie. Vous prendrez soin de dupliquer les chaînes de caractères `chaussure` et `date`. Prototype :

```
Sortie * creer_sortie( char *date,  char *chaussure,  float dist);
```

Solution:

```
Sortie * creer_sortie( char *date,  char *chaussure,  float dist){
    Sortie *nouvelleSortie = (Sortie *)malloc(sizeof(Sortie));
    nouvelleSortie->dist = dist;
    nouvelleSortie->date = strdup(date);// Allouer de la mémoire
       pour la date
    nouvelleSortie->chaussure = strdup(chaussure); // Allouer de la
       mémoire pour la chaussure et copier la valeur
    nouvelleSortie->suiv = NULL;// Initialiser le pointeur suivant à
        NULL
    return nouvelleSortie;
}
```

Question 2 (3 points)

Écrivez une fonction permettant d'insérer une Sortie en fin de liste. La nouvelle sortie (pointée par `nouv`) devra donc se trouver à la fin de la liste spécifiée dans `liste`. La fonction renvoie la liste (lorsque la liste `liste` est vide, la fonction se contentera de renvoyer `nouv`). Prototype :

```
Sortie *inserer_fin(Sortie *liste, Sortie *nouv);
```

Solution:

```
Sortie *inserer_fin(Sortie *liste, Sortie *nouv) {
    // Si la liste est vide, le nouvel élément devient la liste
    if (liste == NULL) {
        return nouv;
    }

    // Sinon, on parcourt la liste jusqu'à la fin
```

```
    Sortie *courant = liste;
    while (courant->suiv != NULL) {
        courant = courant->suiv;
    }

    // On ajoute le nouvel élément à la fin
    courant->suiv = nouv;

    return liste;
}
```

Question 3 (3 points)

Écrivez la fonction permettant de libérer l'intégralité de la mémoire allouée à une liste de sorties de course à pied. Prototype :

```
void liberer(Sortie *liste);
```

Solution:

```
void liberer(Sortie *liste) {
    while (liste) {
        Sortie *suiv=liste->suiv;
        free(liste->date);// Libérer la mémoire allouée pour la chaî
           ne de caractères 'date'
        free(liste->chaussure);// Libérer la mémoire allouée pour la
            chaîne de caractères 'chaussure'
        free(liste);// Enfin, libérer la structure Sortie actuelle
        liste = suiv;
    }
}
```

Question 4 (6 points)

Écrivez une fonction permettant d'afficher un récapitulatif des sorties effectuées avec une paire de chaussures. Si le paramètre `chaussure` est `NULL`, toutes les sorties de toutes les paires sont affichées. La fonction va parcourir les informations relatives à chaque sortie et compte le nombre de kilométrage parcouru avec la paire. Exemple d'affichage si une paire est passée en paramètre :

```
Sortie du 2022-05-07 :  24.340 km avec les Nike Zoom.
Sortie du 2021-08-28 :  3.480 km avec les Nike Zoom.
....
Nombre total de kilomètres parcourus : 807.200 km
```

Prototype :

```
void affiche_liste_sortie(Sortie *liste, char *chaussure);
```

Solution:

```
void affiche_liste_sortie(Sortie *liste, char *chaussure) {
    float totalKilometres = 0.0;
```

```
    // Parcourir la liste des sorties
    while (liste != NULL) {
        if(chaussure == NULL){
           totalKilometres += liste->dist; // Ajouter le kilométrage
               de cette sortie au total
           printf("Sortie␣du␣%s␣:␣␣%.3f␣km␣avec␣les␣%s.\n", liste->
              date, liste->dist, liste->chaussure);
        }
        else if (strcmp(liste->chaussure, chaussure) == 0) {// Vé
           rifier si la paire de chaussures de la sortie correspond
           à la paire spécifiée

            totalKilometres += liste->dist; // Ajouter le kilomé
               trage de cette sortie au total
            printf("Sortie␣du␣%s␣:␣␣%.3f␣km␣avec␣les␣%s.\n", liste->
               date, liste->dist, liste->chaussure);// Afficher les
               informations de la sortie
        }
        liste = liste->suiv;
    }
   printf("Nombre␣total␣de␣kilomètres␣parcourus␣:␣%.3f␣km\n",
      totalKilometres);
}
```

Question 5 (6 points)

Écrivez la fonction de lecture d'un ensemble de sorties depuis un fichier dont le nom est donné en argument à la fonction. Pour chaque Sortie, la première ligne contient la date, la deuxième la paire de chaussures utilisée, la troisième ligne indique la distance en kilomètres de la sortie. Vous prendrez soin de libérer toute la mémoire allouée pour des variables intermédiaires à la fin de la fonction.

```
date
chaussure
distance
```

Exemple de contenu de fichier contenant deux paires de chaussures :

```
2023-11-02
Nike Air
11.7
2023-11-11
Adidas Boston
16.2
```

Prototype :

```
Sortie *lire(const char *fichier);
```

La fonction renvoie la liste des chaussures lues dans le fichier. On supposera que les lignes font au maximum 255 caractères.

Vous pourrez utiliser l'instruction `buffer[strlen(buffer)-1]='\0';` pour enlever le retour à la ligne à la fin de chaque ligne lue.

Solution:

```
Sortie *lire(const char *fichier) {
    FILE *f = fopen(fichier, "r");

    char buffer[256];
    Sortie *listeSorties = NULL;

    while (fgets(buffer, 256, f)) {
        buffer[strlen(buffer)-1]='\0'; /* pour enlever le retour à
           la ligne*/
        char *date=strdup(buffer);

        fgets(buffer, 256, f);
        buffer[strlen(buffer)-1]='\0'; /* pour enlever le retour à
           la ligne*/
        char *chaussure=strdup(buffer);

        fgets(buffer, 256, f);
        float dist;
        sscanf(buffer, "%g", &dist);

        Sortie *nouvelleSortie = creer_sortie(date, chaussure, dist)
           ;
        if (nouvelleSortie != NULL) {
            listeSorties = inserer_fin(listeSorties, nouvelleSortie)
               ;
        }
        free(date);
        free(chaussure);
    }
    fclose(f);
    return listeSorties;
}
```

2 Gestion des sorties liées à chaque paires de chaussures

Au fur et à mesure de l'utilisation de cette structure de données, Léa se rend compte qu'il serait plus simple pour elle d'avoir une structure liée à chaque paire de chaussures qui lui permet de récapituler directement les informations d'une paire. Elle créé donc une structure Chaussure dans laquelle elle souhaite stocker le nombre total de kilomètres réalisés avec une paire, le nombre total et la liste des Sorties qui concerne la paire.

Afin de pouvoir retrouver plus rapidement la paire de chaussures qui l'intéresse, Léa décide de stocker les paires de chaussures dans un arbre binaire de recherche, trié par ordre alphabétique du nom des paires.

Structure :

```
typedef struct _chaussure{
        char* nom;
        float km;
        int nb_sorties;
        Sortie *liste_sorties;
        struct _chaussure *g;
        struct _chaussure *d;
} Chaussure;
```

Le champ `chaussure` devient donc obsolète dans la structure `Sortie`, mais dans un souci de simplicité, nous le gardons ici. La fonction renvoie l'arbre résultant.

Question 6 (6 points)

Ecrire la fonction de l'ajout d'une paire de chaussures dans un arbre binaire de recherche trié par ordre alphabétique.

Prototype :

```
Chaussure *ajout_paire(Chaussure *abr, char *ch);
```

Solution:

```
Chaussure *ajout_paire(Chaussure *abr, char *chaussure) {
    if (!abr) {
        // Si l'arbre est vide, créer une nouvelle paire de
           chaussure
        return creer_chaussure(chaussure);
    }
    int cmp = strcmp(chaussure, abr->nom);
    if (cmp < 0) {
        // La nouvelle chaussure doit etre insérée dans le sous-
           arbre gauche
        abr->g = ajout_paire(abr->g, chaussure);
    } else if (cmp > 0) {
        // La nouvelle chaussure doit etre insérée dans le sous-
           arbre droit
        abr->d = ajout_paire(abr->d, chaussure);
    }
    // Si cmp == 0, la chaussure existe déjà dans l'arbre, aucune
       action n'est nécessaire.
    return abr;
}
```

Question 7 (6 points)

Ecrire la fonction d'ajout des sorties liées à une paire de chaussures. Cette fonction prends en paramètre un pointeur vers le noeud de la chaussure `chaussure` dont on souhaite faire la mise à jour des sorties et un pointeur vers la liste contenant toutes les sorties `liste_sortie`. Attention, on ne souhaite ajouter à la chaussure que les sorties qui ont été réalisées avec cette paire.

La fonction prendra soin de dupliquer la sortie correspondante et l'inserera dans la liste des sorties de la chaussure. La fonction mettra également à jour les champs `km` et `nb_sorties` en fonction de la sortie ajoutée.

La fonction de duplication d'une `Sortie` vous est fournie et a le prototype suivant :

```
Sortie *dupliquer(Sortie *Sortie);
```

Prototype :

```
void ajout_sorties(Chaussure *chaussure, Sortie *liste_sortie);
```

Solution:

```
void ajout_sorties(Chaussure *chaussure, Sortie *liste_sortie) {
    // Vérification des arguments d'entrée
    if (chaussure == NULL || liste_sortie == NULL) {
        return;
    }
    Sortie * sortie_tmp;
    // Parcourir liste_sortie
    while (liste_sortie != NULL) {

        // Vérifier que la chaussure de la sortie correspond à la
           chaussure passée en paramètres
        if (strcmp(chaussure->nom, liste_sortie->chaussure) == 0) {
            sortie_tmp=dupliquer(liste_sortie);
            chaussure->liste_sorties = inserer_fin(chaussure->
               liste_sorties, sortie_tmp);
            chaussure->km += liste_sortie->dist;
            chaussure->nb_sorties++;
        }
        liste_sortie = liste_sortie->suiv;
    }
}
```

Question 8 (3 points)

Ecrire la fonction de recherche d'une paire de chaussures dans un arbre. La fonction prend en paramètre l'arbre `abr` dans lequel effectuer la recherche ainsi que le `nom` de la paire de chaussures à rechercher et renvoie un pointeur vers la paire de chaussures si elle a été trouvée, sinon la fonction renvoie `NULL`.

Prototype

```
Chaussure *rechercher_abr(Chaussure *abr, const char *nom);
```

Solution:

```
Chaussure *rechercher_abr(Chaussure *abr, const char *nom)
{
        if (abr == NULL) return NULL;
        if (strcmp(nom, abr->nom) == 0) return abr;
        if (strcmp(nom, abr->nom) < 0) return rechercher_abr(abr->g,
            nom);
        return rechercher_abr(abr->d, nom);
}
```

Question 9 (3 points)

Maxime, un ami de Léa, souhaite l'aider et lui propose de coder une fonction qui parcours l'arbre des chaussure et qui écrit le nom des paires de chaussures ayant parcouru plus de 1000km, ce qui signifie qu'elle est sûrement trop usée. Voici le code qu'il a écrit.

```
204 void chaussure_usee(Chaussure *abr){
205         if(abr->km > 1000) printf("La␣paire␣%s␣est␣usée,␣elle␣a␣parcourue␣plus␣de␣
                %.3f␣km␣\n", abr->nom, abr->km);
206         chaussure_usee(abr->g);
207         chaussure_usee(abr->d);
208 }
```

Cependant, à l'utilisation, la fonction renvoie une erreur de segmentation. Il execute donc son code avec valgrind afin de trouver son erreur. Voici la sortie de valgrind :

```
==8312== Invalid read of size 4
==8312==    at 0x109D1F: chaussure_usee (sortie_bug.c:205)
==8312==    by 0x109D75: chaussure_usee (sortie_bug.c:206)
==8312==    by 0x109359: main (test.c:18)
==8312==  Address 0x8 is not stack'd, malloc'd or (recently) free'd
==8312==
==8312==
==8312== Process terminating with default action of signal 11 (SIGSEGV)
==8312==  Access not within mapped region at address 0x8
==8312==    at 0x109D1F: chaussure_usee (sortie_bug.c:205)
==8312==    by 0x109D75: chaussure_usee (sortie_bug.c:206)
==8312==    by 0x109359: main (test.c:18)
==8312==  If you believe this happened as a result of a stack
==8312==  overflow in your program's main thread (unlikely but
==8312==  possible), you can try to increase the size of the
==8312==  main thread stack using the --main-stacksize= flag.
==8312==  The main thread stack size used in this run was 8388608.
==8312==
==8312== HEAP SUMMARY:
==8312==     in use at exit: 12,145 bytes in 607 blocks
==8312==   total heap usage: 809 allocs, 202 frees, 18,996 bytes allocated
==8312==
==8312== LEAK SUMMARY:
==8312==    definitely lost: 0 bytes in 0 blocks
==8312==    indirectly lost: 0 bytes in 0 blocks
==8312==      possibly lost: 0 bytes in 0 blocks
==8312==    still reachable: 12,145 bytes in 607 blocks
==8312==         suppressed: 0 bytes in 0 blocks
==8312== Rerun with --leak-check=full to see details of leaked memory
==8312==
==8312== For lists of detected and suppressed errors, rerun with: -s
==8312== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Erreur de segmentation (core dumped)
```

Indiquez où se trouvent la ou les erreurs, à quoi sont elles-dûes ? Que faut il faire pour corriger le problème ?

Solution: L'erreur est générée par la ligne 205. Cette ligne se trouve dans la fonction `chaussure_usee` qui a été appellée de manière récursive, d'abord dans la fonction `chaussure_usee` elle même ligne 206 puis dans le main à la ligne 18. En effet, Il n'y a pas de condition d'arrêt à cette fonction recursive (il n'y a jamais de return pour stopper la fonction). Lorsqu'on atteind l'extrémtité de l'arbre, les fils gauche et droits sont nuls, les lignes 206 et 207 font donc des appels récursifs sur des valeurs vides. Lorsque le nouvel appel à la fonction s'execute et cherche a atteindre le champ `km` d'une structure non allouée, cela créé une erreur de segmentation mémoire. Afin de corriger cela il faudrait rajouter une condition d'arrêt à la fonction, en ajoutant au début `if(!abr) return;`

Question 10 (6 points)

Léa souhaiterait pouvoir trier les sorties liées à une paire de chaussures par distance croissante. Pour cela elle souhaite pouvoir utiliser un appel à la fonction `qsort` dont le prototype est le suivant :

```
void qsort(void *base, size_t nel, size_t width, int (*compar)(const
   void *, const void *));
```

Ecrivez la fonction de comparaison `compare_sorties` permettant de trier un tableau de `Sortie` par ordre croissant des distances. Puis écrivez une fonction `trier_sorties` qui renverra le tableau de pointeur de sorties triées pour une chaussure donnée (passée en argument de la fonction). Cette fonction devra, à partir de la liste des sorties de la chaussure, allouer la mémoire pour un tableau de pointeurs de la taille du nombre de sortie, puis le remplir avec les pointeurs sur les sorties de la chaussure avant de trier ce tableau de pointeur de `Sortie` par ordre croissant des distances et de le renvoyer.

Prototypes :

```
int compare_sorties(const void *a, const void *b);

Sortie** trier_sorties(Chaussure *chaussure);
```

Solution:

```
int compare_sorties(const void *a, const void *b) {
    // Convertir les pointeurs génériques en pointeurs de Sortie
    const Sortie *sortieA = *(const Sortie **)a;
    const Sortie *sortieB = *(const Sortie **)b;

    // Comparer les distances
    if (sortieA->dist < sortieB->dist) return -1;
    if (sortieA->dist > sortieB->dist) return 1;

    return 0; // Les distances sont égales
}

// Fonction pour trier les sorties d'une paire de chaussures par
   nombre de km
Sortie ** trier_sorties(Chaussure *chaussure) {
    // Vérifier si la chaussure est valide
    if (chaussure == NULL) {
        printf("Chaussure␣invalide.\n");
        return NULL;
    }

    // Vérifier si la liste des sorties de la chaussure est vide
    if (chaussure->liste_sorties == NULL) {
        printf("Aucune␣sortie␣à␣trier␣pour␣la␣chaussure␣%s.\n",
           chaussure->nom);
        return NULL;
    }

    int nbSorties = chaussure->nb_sorties;
    // Allouer un tableau de pointeurs vers les sorties
    Sortie **tableauSorties = malloc(nbSorties * sizeof(Sortie *));
```

```
    if (tableauSorties == NULL) {
        printf("Erreur␣d'allocation␣mémoire.\n");
        return NULL;
    }

    // Remplir le tableau de pointeurs
    int i = 0;
    Sortie* listeSorties = chaussure->liste_sorties;
    while (listeSorties != NULL) {
        tableauSorties[i] = listeSorties;
        listeSorties = listeSorties->suiv;
        i++;
    }

    // Utiliser qsort pour trier le tableau de pointeurs
    qsort(tableauSorties, nbSorties, sizeof(Sortie *),
       compare_sorties);

    return tableauSorties;

}
```

Question 11 (6 points)

Ecrivez un main permettant, en s'appuyant sur les fonctions vues précédemment, de charger la liste des sorties stockée dans le fichier `sorties.txt`, d'afficher les sorties réalisées avec la paire `Adidas Boost` puis de créer un arbre avec toutes les paires de chaussures différentes qui existent dans la liste. On affichera ensuite toutes les paires de chaussures usées (qui ont parcourue plus de 1000km). Enfin, on recherchera la chaussure "New Balance" et on triera ses sorties par ordre croissant des distances. Puis on affichera les sorties triées dans l'ordre croissant. On terminera par libérer toute la mémoire allouée par le programme. Vous disposez de la fonction `liberer_abr` qui permet de libérer la mémoire allouée pour un arbre. Son prototype est :

```
void liberer_abr(Chaussure *abr);
```

Vous disposez également de la fonction `creer_abr` qui permet de créer un arbre à partir d'une liste de sortie, et d'associer à chaque paire de chaussure la liste de sortie qui lui est liée. Son prototype est :

```
Chaussure *creer_abr(Sortie *liste_sorties);
```

Solution:

```
#include "sortie.h"
#include<stdio.h>
#include <stdlib.h>
#include<assert.h>

int main() {
```

```
    // Lecture depuis le fichier
    Sortie *nouvelleListe = lire("sorties.txt");

    // Affichage de la nouvelle liste des sorties
    affiche_liste_sortie(nouvelleListe, NULL);
    affiche_liste_sortie(nouvelleListe, "Adidas Boost");

    printf("\n Création d'un arbre de chaussures \n");
    Chaussure* abr = creer_abr(nouvelleListe);

    chaussure_usee(abr);

    Chaussure *ch = rechercher_abr(abr, "New Balance");
    Sortie **tableauSorties=trier_sorties(ch);
     // Afficher les sorties triées
    printf("Sorties triées pour la chaussure %s :\n", ch->nom);
    for (int j = 0; j < ch->nb_sorties; j++) {
        printf("Sortie du %s : %.3f km\n", tableauSorties[j]->date,
           tableauSorties[j]->dist);
    }
    free(tableauSorties);
    liberer_abr(abr);
    liberer(nouvelleListe);

    return 0;
}
```

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

========== Fichier `sortie.h` ==========

```
#ifndef _SORTIE_H_
#define _SORTIE_H_

typedef struct _sortie {
    char* date;
    char *chaussure; /* texte décrivant la marque et le modèle de la
        paire de chaussure */
    float dist; /* nombre de kilomètres de la sortie en km */
    struct _sortie *suiv;
} Sortie;

/* Q1 Création d'une Sortie */
Sortie * creer_sortie( char *date,  char *chaussure,  float dist);

/* Q2 Insertion dans une liste */
Sortie *inserer_fin(Sortie *liste, Sortie *nouv);

/* Q3 libération d'une liste */
void liberer(Sortie *liste);

/* Q4 les sorties liées à une paire de chaussure */
void affiche_liste_sortie(Sortie *liste, char *chaussure);

/* écriture dans un fichier */
void ecrire(Sortie *liste, const char *fichier) ;

/* Q5 lire depuis un fichier */
Sortie *lire(const char *fichier);

/* Dupliquer une Sortie (seule)*/
Sortie *dupliquer(Sortie *Sortie);

/* Structure permettant de stocker les paires de chaussures */
typedef struct _chaussure{
        char* nom;
        float km;
        int nb_sorties;
        Sortie *liste_sorties;
        struct _chaussure *g;
        struct _chaussure *d;
} Chaussure;

/* Creation d'une paire de chaussure */
Chaussure *creer_chaussure(char *chaussure);

/* Q6 Ajout d'une paire de chaussure dans un arbre binaire trié par
    ordre alphabétique */
Chaussure *ajout_paire(Chaussure *abr, char *ch);

/* Q7 Ajout d'une liste de sorties pour une chaussure en particulier */
void ajout_sorties(Chaussure *chaussure, Sortie *liste_sortie);

/* Q8 Rechercher dans un arbre si un paire de chaussure est présente*/
Chaussure *rechercher_abr(Chaussure *abr, const char *nom);

/* Q9 imprime toutes les chaussures usées*/
void chaussure_usee(Chaussure *abr);

/* Q10 Trie d'un tableau de sortie avec la fonction qsort */
int compare_sorties(const void *a, const void *b);
Sortie** trier_sorties(Chaussure *chaussure);

/* Création d'un abre à partir d'une liste de sorties */
Chaussure *creer_abr(Sortie *liste_sorties);

/* Libérer la mémoire allouée pour un arbre */
void liberer_abr(Chaussure *abr);

#endif
```