SCIENCES SORBONNE UNIVERSITÉ

Votre numéro d'anonymat : | | | |

Programmation et structures de données en C– LU2IN018

Examen du 14 janvier 2022
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 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.

Sciences participatives et observations de plantes

1 Gestion des observations avec des listes

Les smartphones permettent de prendre en photos et de filmer en pleine nature, d'associer ces observations à une position GPS et d'envoyer ces informations à une plateforme de recueil de données via une application dédiée pour les partager avec une communauté de personnes intéressées et avec des scientifiques qui peuvent ensuite en tirer des informations très utiles pour suivre l'évolution de populations animales ou de plantes. Vous allez écrire du code permettant de gérer des observations de plantes (sans photo ni video, juste l'indication qu'une plante particulière a été vue à une position GPS donnée). Les observations de plantes permettront d'associer un nom et un descriptif à une position GPS. Structures :

```
typedef struct _plante {
  char *nom;
  char *descriptif;
} Plante;
```

```
typedef struct _gps_pos {
  double latitude;
  double longitude;
} GPS_Pos;
```

```
typedef struct _observation {
  Plante plante;
  GPS_Pos pos;
} Observation;
```

Question 1 (3 points)

Écrivez la fonction de création d'une observation. Le nom et le descriptif doivent être recopiés.

Prototype :

```
Observation *creer_observation(const char *nom, const char *
   descriptif, GPS_Pos pos);
```

Réponse :

Les observations seront dans un premier temps stockées dans une liste chaînée.

Structure :

```
typedef struct _elt_observation {
  Observation *obs;
  struct _elt_observation *suiv;
} Elt_Observation;
```

Question 2 (3 points)

Écrivez la fonction d'ajout d'une observation à une liste. Prototype :

```
void ajouter_observation(Elt_Observation **plobs, Observation *obs);
```

L'argument `plobs` contient la liste à laquelle ajouter l'observation (passage par pointeur). L'observation n'aura pas besoin d'être dupliquée.

L'ordre des observations dans la liste n'ayant pas d'importance, vous choisirez d'effectuer l'insertion à l'endroit le plus pertinent afin de minimiser les opérations à effectuer lors de l'ajout d'une nouvelle observation. Par contre, comme pour les autres questions, votre fonction devra respecter le prototype indiqué.

Réponse :

Question 3 (3 points)

Écrivez la fonction d'écriture d'une observation dans un fichier (déjà ouvert). Prototype :

```
void ecrire_observation(Observation *obs, FILE *f);
```

Exemple :

```
Plante: Grande Astrance
Descriptif: Herbacée vivace de la famille des Apiaceae
Latitude: 45.641, longitude: 6.582
```

Réponse :

Question 4 (6 points)

Écrivez les deux fonctions permettant de libérer la mémoire associée à une liste d'observations. La première libère une seule observation (et toute la mémoire associée) et la seconde s'appuie sur celle-ci pour libérer une liste chaînée d'observations.

Prototypes :

```
void liberer_observation(Observation *obs);
```

```
void liberer_liste_observations(Elt_Observation *lobs, int
   liberer_obs);
```

L'argument `liberer_obs` indiquera si l'observation portée par un élément de la liste doit être supprimée (`liberer_obs=1`) ou non (`0`).

Réponse :

2 Gestion des observations avec des arbres

Pour pouvoir retrouver plus efficacement des observations, nous allons les stocker dans un arbre de type quad-tree. Il s'agit d'un arbre avec quatre fils : nord-ouest, nord-est, sud-ouest, sud-est. Chaque noeud de l'arbre contient une observation (et une seule). Le sous-arbre nord-est contient toutes les observations qui sont au nord-est de l'observation portée par le noeud courant. Le sous-arbre nord-ouest contient toutes les observations qui sont au nord-ouest du noeud courant, etc [1].

1. C'est comme un arbre binaire de recherche, sauf qu'il y a 4 sous-arbres au lieu de 2 pour pouvoir stocker des points en 2 dimensions, pas seulement à gauche et à droite du point courant, mais aussi au-dessus et en-dessous.

Structure :

```
typedef struct _quad_tree {
  Observation *obs;
  struct _quad_tree *no;
  struct _quad_tree *ne;
  struct _quad_tree *se;
  struct _quad_tree *so;
} Quad_Tree;
```

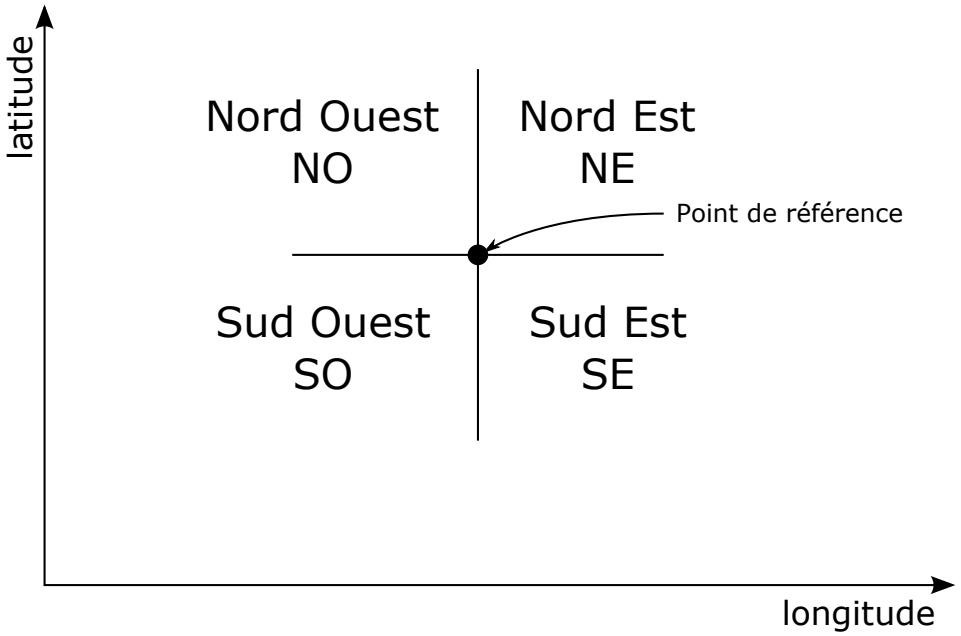

FIGURE 1 – Découpage de l'espace utilisé dans un quad-tree. Chaque sous-arbre contient des points qui sont, respectivement, dans le quadrant nord-ouest, nord-est, sud-ouest ou sud-est.

Question 5 (6 points)

Écrivez la fonction d'ajout d'une observation dans un quad-tree. Prototype :

```
Quad_Tree *ajout_QT(Quad_Tree *qt, Observation *obs);
```

Les coordonnées seront considérées comme toujours différentes, il ne sera pas nécessaire de gérer les égalités.

La fonction fera un ajout dans l'arbre de façon récursive. Pour simplifier l'écriture, il est recommandé de mettre les latitudes et longitudes dans des variables avec un nom court. Vous commenterez le code pour indiquer à quelle région un bloc d'instructions correspond.

Réponse :

Question 6 (3 points)

Écrivez la fonction d'écriture dans un fichier des observations stockées dans un arbre QT en préfixe : écriture de l'observation portée par le noeud, puis écriture des différents sous-arbres dans le même ordre que déclarés dans la structure. Prototype :

```
void ecrire_QT(Quad_Tree *qt, const char *fichier);
```

Vous utiliserez une fonction récursive (que vous devrez écrire). Prototype :

```
void ecrire_QT_rec(Quad_Tree *qt, FILE *f);
```

Vous vous appuierez sur la fonction d'écriture d'une observation vue précédemment.

Réponse :

Question 7 (6 points)

La fonction de lecture a été écrite et testée avec le main indiqué ci-après qui lit les observations qui sont dans le fichier `observations_2.txt` et écrit dans le fichier `observations_3.txt` avant de libérer la mémoire allouée à l'arbre.

```
77 Observation *lire_observation(FILE *f) {
78   char buff[300];
79   if (fgets(buff, 300, f)==NULL) return NULL;
80   buff[strlen(buff)-1]='\0';
81   char *nom=strdup(buff+8);
82   fgets(buff, 300, f);
83   buff[strlen(buff)-1]='\0';
84   char *descriptif=strdup(buff+12);
85   GPS_Pos pos;
86   fgets(buff, 300, f);
87   sscanf(buff, "Latitude: %lf, longitude: %lf", &pos.latitude, &pos.
       longitude);
```

```
88   Observation *obs=creer_observation(nom, descriptif, pos);
89   return obs;
90 }
91
92 Quad_Tree *lire_QT(const char *fichier) {
93   FILE *f=fopen(fichier, "r");
94   Observation *obs=lire_observation(f);
95   Quad_Tree *qt=NULL;
96   while(obs) {
97     qt=ajout_QT(qt, obs);
98     obs=lire_observation(f);
99   }
100  fclose(f);
101  return qt;
102 }
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include "observation.h"
5
6  int main(void) {
7    srand(time(NULL));
8    Quad_Tree *qt=lire_QT("observations_2.txt");
9    ecrire_QT(qt, "observations_3.txt");
10   liberer_QT(qt);
11   return 0;
12 }
```

L'exécution de valgrind sur ce programme donne les messages suivants :

```
valgrind --leak-check=full ./main_obs4
==2788== Memcheck, a memory error detector
==2788== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2788== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2788== Command: ./main_obs4
==2788==
==2788==
==2788== HEAP SUMMARY:
==2788==     in use at exit: 14,000 bytes in 2,000 blocks
==2788==   total heap usage: 7,004 allocs, 5,004 frees, 125,488 bytes allocated
==2788==
==2788== 7,000 bytes in 1,000 blocks are definitely lost in loss record 1 of 2
==2788==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/...)
==2788==    by 0x4ED9A29: strdup (strdup.c:42)
==2788==    by 0x108FF2: lire_observation (observation.c:81)
==2788==    by 0x10911B: lire_QT (observation.c:94)
==2788==    by 0x108B6E: main (main_obs4.c:9)
==2788==
```

```
==2788== 7,000 bytes in 1,000 blocks are definitely lost in loss record 2 of 2
==2788==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/...)
==2788==    by 0x4ED9A29: strdup (strdup.c:42)
==2788==    by 0x109042: lire_observation (observation.c:84)
==2788==    by 0x10911B: lire_QT (observation.c:94)
==2788==    by 0x108B6E: main (main_obs4.c:9)
==2788==
==2788== LEAK SUMMARY:
==2788==    definitely lost: 14,000 bytes in 2,000 blocks
==2788==    indirectly lost: 0 bytes in 0 blocks
==2788==      possibly lost: 0 bytes in 0 blocks
==2788==    still reachable: 0 bytes in 0 blocks
==2788==         suppressed: 0 bytes in 0 blocks
==2788==
==2788== For counts of detected and suppressed errors, rerun with: -v
==2788== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
```

Quel est le problème ? Vous justifierez votre réponse sur la base des messages de valgrind en indiquant notamment les numéros des lignes incriminées. Proposez une solution.

Réponse :

Question 8 (6 points)

Écrivez la fonction de recherche des observations situées dans une zone donnée. Prototype :

```
Elt_Observation *chercher_observation(Quad_Tree *qt, GPS_Pos pso,
   GPS_Pos pne);
```

Le premier argument indique l'arbre dans lequel faire la recherche. L'argument `pso` donne la limite sud-ouest de la zone, l'argument `pne` la limite nord-est de la zone. Cela signifie que les coordonnées à afficher sont celles pour lesquelles la latitude est entre la latitude de `pso` (qui est la plus petite) et la latitude de `pne` et la longitude est entre la longitude de `pso` (qui est également la plus petite) et la longitude de `pne` [2] (zone 9 de la figure 2).

Cette fonction nécessite de répéter le même type d'opérations pour plusieurs zones. Vous pourrez ne détailler que les cas suivants : point au sud-ouest de `pso` (zone numérotée 1 sur la figure 2), point de

2. La bordure est considérée comme faisant partie de la zone.

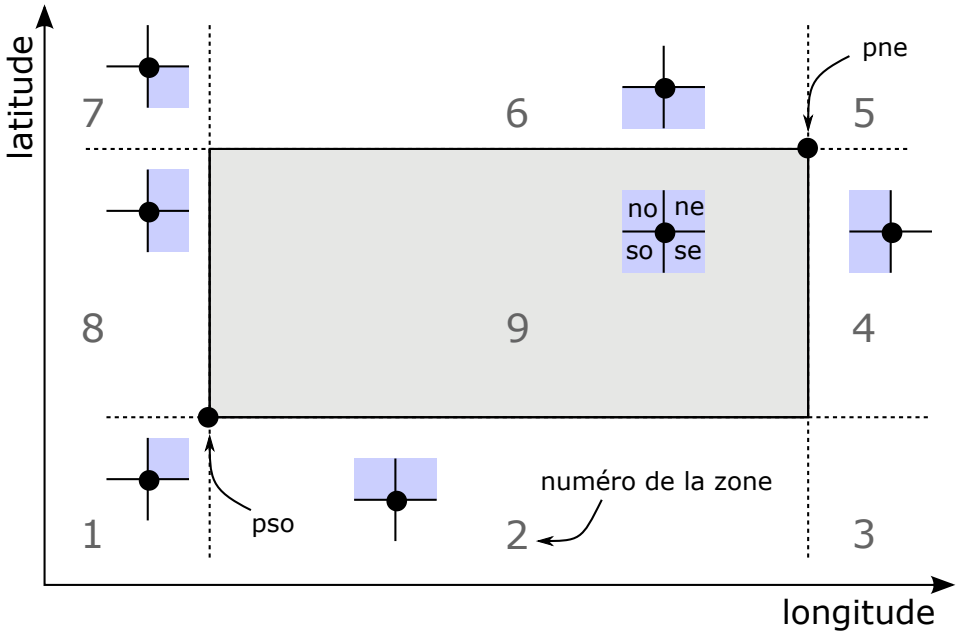

FIGURE 2 – Quadrants à considérer dans la fonction de recherche en fonction de la position du point par rapport aux points délimitant la zone de recherche. Les quadrants non grisés n'ont pas besoin d'être considérés dans la recherche. Par exemple, seul le quadrant nord-est doit être exploré à partir d'un point appartenant à la zone 1 pour trouver de potentiels points appartenant à la zone de recherche.

latitude inférieure à `pso` et de longitude intermédiaire entre `pso` et `pne` (zone 2) et point dans la zone recherchée (zone 9).
La valeur de retour est un pointeur sur le premier élément d'une liste que vous aurez construite et qui contient toutes les observations trouvées dans la zone de recherche. Vous pourrez utiliser la fonction de concaténation de listes suivante (vous n'aurez pas à l'écrire) :

```
Elt_Observation *concatener_listes(Elt_Observation *lobs1,
   Elt_Observation *lobs2);
```

La fonction ajoute la deuxième liste à la fin de la première (si cette dernière existe). Elle renvoie le pointeur vers le premier élément de la liste résultant de la concaténation.

Réponse :

Réponse :

Question 9 (6 points)

Écrivez une fonction `main` créant 1000 observations aléatoires avec l'aide de la fonction `random_obs` entre les coordonnées (43.3, -4.31) et (50.38, 7.75)[3]. Les observations seront ajoutées à un arbre QT, l'arbre sera écrit dans le fichier "observation.txt", puis la liste des observations qui sont entre les coordonnées (44,33, 6.05) et (45.55 et 6.51) seront affichées avant de libérer la mémoire.

Prototype :

```
Observation *random_obs(GPS_Pos so, GPS_Pos ne);
```

Remarques :

— l'utilisation de `random_obs` nécessite d'initialiser le générateur pseudo-aléatoire

— l'affichage à l'écran peut se faire avec les fonctions d'écriture dans un fichier en utilisant la variable `stdout`, qui est un `FILE *` qui permet d'accéder au terminal comme à un fichier (`fprintf(stdout, "message")` est équivalent à `printf("message")`).

3. Les coordonnées sont données sous la forme (latitude, longitude). Il s'agit des coordonnées qui encadrent grossièrement la France métropolitaine, avec la latitude de Bayonne et la longitude de Brest d'un côté et la latitude de Lille et la longitude de Strasbourg de l'autre.

Réponse :

3 Listes génériques

Dans la suite, il sera nécessaire de manipuler différents types de listes (de plantes, de photos, ...). Dans ces conditions, pour préparer ces futurs développements et pour éviter de dupliquer du code, il vous est demandé de basculer les listes d'observations vers des listes génériques.

Structures :

```
typedef struct _element *PElement;
typedef struct _element {
  void *data;
  PElement suivant;
} Element;

typedef struct _liste *PListe;
```

```
typedef struct _liste {
  PElement elements;
  void *(*dupliquer)(const void *src);
  void (*copier)(const void *src, void *dst);
  void (*detruire)(void *data);
  void (*afficher)(const void *data);
  int (*comparer)(const void *a, const void *b);
  void (*ecrire)(const void *data, FILE *f);
  void *(*lire)(FILE *f);
} Liste;
```

Question 10 (6 points)

Écrivez les fonctions suivantes pour manipulation des observations en vue de leur utilisation avec la bibliothèque générique de listes.

```
void *dupliquer_obs(const void *src);

void copier_obs(const void *src, void *dst);

void detruire_obs(void *data);
```

Pour rappel, la fonction `dupliquer_obs` alloue une nouvelle observation et copie `src` dans cette nouvelle structure. `copier_obs` recopie le contenu d'une observation `src` dans une observation `dst` déjà existante auparavant et `detruire_obs` libère toute la mémoire associée à une observation.

Remarques :

— Vous pourrez vous appuyez sur les fonctions définies précédemment

— Vous prendrez garde à ce qu'aucune de ces fonctions ne crée de fuite mémoire. Lors de la copie vers `dst`, vous prendrez ainsi garde à ce que les chaînes du nom et du descriptif soient à la bonne taille.

Réponse :

Réponse :

Les autres fonctions nécessaires seront supposées disponibles sans avoir besoin de les écrire.

Question 11 (3 points)

Écrivez une fonction `main` pour créer une liste générique d'observations, y insérer 10 observations générées aléatoirement avant d'afficher la liste. Vous prendrez soin de bien libérer toute la mémoire allouée. Les noms des fichiers en-tête sont indiqués dans la page détachable à la fin de l'énoncé.

Vous pourrez utiliser les fonctions suivantes :

```
Liste *creer_liste(
                    void *(*dupliquer)(const void *src),
                    void (*copier)(const void *src, void *dst),
                    void (*detruire)(void *data),
                    void (*afficher)(const void *data),
                    int (*comparer)(const void *a, const void *b),
                    void (*ecrire)(const void *data, FILE *f),
                    void *(*lire)(FILE *f)
                    );

void detruire_liste(PListe pliste);

void afficher_liste(PListe pliste);

void inserer_debut(PListe pliste, void *data);
```

Remarque : la fonction `inserer_debut` duplique la donnée et la fonction `detruire_liste` la détruit.

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

========== Fichier observation.h ==========

```
typedef struct _plante {
  char *nom;
  char *descriptif;
} Plante;

typedef struct _gps_pos {
  double latitude;
  double longitude;
} GPS_Pos;

typedef struct _observation {
  Plante plante;
  GPS_Pos pos;
} Observation;

typedef struct _quad_tree {
  Observation *obs;
  struct _quad_tree *no;
  struct _quad_tree *ne;
  struct _quad_tree *se;
  struct _quad_tree *so;
} Quad_Tree;

/* Création d'une observation */
Observation *creer_observation(const char *nom,
   const char *descriptif, GPS_Pos pos);

/* Ecriture d'une observation */
void ecrire_observation(Observation *obs, FILE *f);

/* Libération d'une observation */
void liberer_observation(Observation *obs);

/* Libération d'une liste d'observations */
void liberer_liste_observations(Elt_Observation *
   lobs, int liberer_obs);

/* Ajout d'une observation dans un quad-tree */
Quad_Tree *ajout_QT(Quad_Tree *qt, Observation *obs)
   ;

/* Ecriture d'un quad-tree dans un fichier */
void ecrire_QT(Quad_Tree *qt, const char *fichier);

/* Lecture d'une observation */
Observation *lire_observation(FILE *f);

/* Lecture d'un quad tree */
Quad_Tree *lire_QT(const char *fichier);

/* Libération d'un quad-tree */
void liberer_QT(Quad_Tree *qt);

/* Ajout d'une observation à une liste d'
   observations */
void ajouter_observation(Elt_Observation **plobs,
   Observation *obs);

/* Concaténation de deux listes d'observations */
Elt_Observation *concatener_listes(Elt_Observation *
   lobs1, Elt_Observation *lobs2);

/* Recherche d'un ensemble d'observations dans une
   zone donnée */
Elt_Observation *chercher_observation(Quad_Tree *qt,
    GPS_Pos pso, GPS_Pos pne);

/* Génération aléatoire d'une observation */
Observation *random_obs(GPS_Pos so, GPS_Pos ne);

/* Affichage d'une liste d'observation */
```

```
void afficher_observations(Elt_Observation *lobs);

/* Fonction récursive d'écriture de l'arbre */
void ecrire_QT_rec(Quad_Tree *qt, FILE *f);
```

========== Fichier `liste_generique.h` ==========

```
typedef struct _element *PElement;
typedef struct _element {
  void *data;
  PElement suivant;
} Element;

typedef struct _liste *PListe;
typedef struct _liste {
  PElement elements;
  void *(*dupliquer)(const void *src);
  void (*copier)(const void *src, void *dst);
  void (*detruire)(void *data);
  void (*afficher)(const void *data);
  int (*comparer)(const void *a, const void *b);
  void (*ecrire)(const void *data, FILE *f);
  void *(*lire)(FILE *f);
} Liste;


Liste *creer_liste(
                   void *(*dupliquer)(const void *
                      src),
                   void (*copier)(const void *src,
                      void *dst),
                   void (*detruire)(void *data),
                   void (*afficher)(const void *data
                      ),
                   int (*comparer)(const void *a,
                      const void *b),
                   void (*ecrire)(const void *data,
                      FILE *f),
                   void *(*lire)(FILE *f)
                   );

/* Insertion au début d'une liste générique */
void inserer_debut(PListe pliste, void *data);

/* Chercher un élément dans une liste générique */
PElement chercher_liste(PListe pliste, void *data);

/* Détruire une liste générique */
void detruire_liste(PListe pliste);

/* Afficher une liste générique */
void afficher_liste(PListe pliste);
```

========== Fichier `fonctions_observation.h` ==========

```
/* Duplication d'une observation */
void *dupliquer_obs(const void *src);

/* Copier une observation (sans allocation) */
void copier_obs(const void *src, void *dst);

/* Destruction d'une observation */
void detruire_obs(void *data);

/* Affichage d'une observation */
void afficher_obs(const void *data);

/* Comparer des observations */
int comparer_obs(const void *a, const void *b);

/* Ecrire une observation */
void ecrire_obs(const void *data, FILE *f);

/* Lire une observation */
void *lire_obs(FILE *f);
```