Votre numéro d'anonymat : | | | |

Programmation et structures de données en C– 2I001

Examen du 16 janvier 2017
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 60 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.

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.

Vélo-Libre

Vous allez développer des fonctions permettant de gérer un système de vélos en "flotte-libre". Ces vélos sont empruntés et déposés par leurs utilisateurs n'importe où, sans base fixe. Ils disposent d'un système permettant de les localiser. Ce sujet va s'intéresser à l'entretien de cette flotte en guidant le personnel chargé de l'entretien des vélos.

Un vélo dispose d'un identifiant, d'une position et d'un état :

```
typedef enum _etat {Neuf=0,Usage, Endommage} Etat;

typedef struct _velo {
  int id;
  Etat etat;
  double x;
  double y;
} Velo;
```

1 QuadTree

Pour pouvoir retrouver rapidement les vélos, ils vont être rangés dans une structure de type QuadTree. Il s'agit d'un arbre quaternaire, qui a donc 4 sous-arbres. Chaque sous-arbre contient les vélos qui sont, respectivement, au nord-ouest, nord-est, sud-ouest ou sud-est du vélo stocké dans la racine (figure 1). La structure de données correspondante est :

```
typedef struct _quadtree QuadTree;
struct _quadtree {
  Velo *velo;
  QuadTree *NW;
  QuadTree *NE;
  QuadTree *SW;
  QuadTree *SE;
};
```

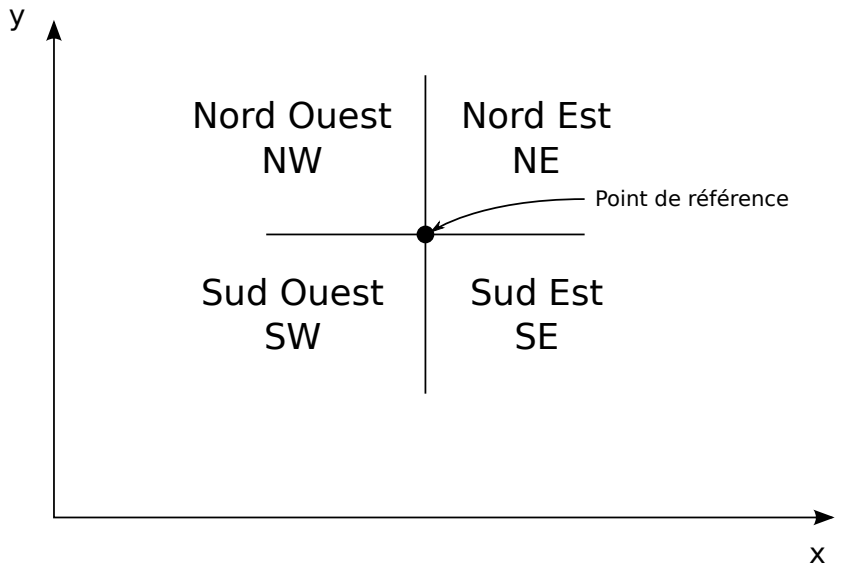

FIGURE 1 – Secteurs associés à une position.

Question 1 (3 points)

Écrivez la fonction de création d'un nœud. Il ne sera pas nécessaire d'allouer une copie du vélo passé en argument :

```
QuadTree *creer_quadtree(Velo *v);
```

Solution:

```
QuadTree *creer_quadtree(Velo *v) {
  QuadTree *qt=malloc(sizeof(QuadTree));
  qt->velo=v;
  qt->NW=NULL;
  qt->NE=NULL;
  qt->SW=NULL;
  qt->SE=NULL;
  return qt;
}
```

Question 2 (6 points)

L'insertion dans un QuadTree se fait comme dans un arbre binaire de recherche. Les coordonnées du vélo à insérer sont comparées aux coordonnées du vélo de la racine de l'arbre. Le vélo sera ajouté au sous-arbre correspondant.

Prototype :

```
QuadTree *inserer(QuadTree *qt, Velo *v);
```

Solution:

```
QuadTree *inserer(QuadTree *qt, Velo *v) {
  if(qt==NULL) return creer_quadtree(v);
  double x1=v->x;
  double y1=v->y;
  double x2=qt->velo->x;
  double y2=qt->velo->y;
  if (x1<x2) {
    if (y1<y2) {
      qt->SW=inserer(qt->SW,v);
    }
    else {
      qt->NW=inserer(qt->NW,v);
    }
  }
  else {
    if (y1<y2) {
      qt->SE=inserer(qt->SE,v);
    }
    else {
      qt->NE=inserer(qt->NE,v);
    }
  }
  return qt;
}
```

Question 3 (6 points)

Écrivez les fonctions permettant d'enregistrer l'arbre dans un fichier. Vous définirez une fonction `sauvegarder` permettant d'ouvrir le fichier et de faire appel à la fonction **récursive** `ecrire_quadtree` qui écrira l'arbre dans le fichier. Une ligne décrira un nœud du QuadTree avec ses 4 champs : le numéro d'identification du vélo, l'entier correspondant à l'état (0, 1 ou 2) et les coordonnées flottantes (le format sera le format par défaut des nombres flottants).

Exemple de contenu (id etat x y) :

```
75926 0 83.527125 40.392861
60408 2 14.732731 13.016975
81109 1 77.103639 80.863986
```

Vous ne gérerez pas des erreurs éventuelles à l'ouverture du fichier. Vous écrirez une fonction récursive pour écrire chacun des nœuds de l'arbre et la fonction `sauvegarder` qui ouvrira le fichier en écriture et fera appel à cette fonction à partir de la racine du QuadTree.

Prototypes :

```
void ecrire_quadtree(QuadTree *qt, FILE *f);
void sauvegarder(QuadTree *qt, const char *fichier);
```

Solution:

```
void ecrire_quadtree(QuadTree *qt, FILE *f) {
  if (qt) {
    fprintf(f,"%d␣%d␣%f␣%f\n", qt->velo->id, qt->velo->etat,
           qt->velo->x, qt->velo->y);
    ecrire_quadtree(qt->NW,f);
    ecrire_quadtree(qt->NE,f);
    ecrire_quadtree(qt->SW,f);
    ecrire_quadtree(qt->SE,f);
  }
}

void sauvegarder(QuadTree *qt, const char *fichier) {
  FILE *f=fopen(fichier,"w");
  ecrire_quadtree(qt,f);
  fclose(f);
}
```

Question 4 (9 points)

Écrivez la fonction permettant de lire les vélos depuis un fichier et de les mettre dans un QuadTree. Prototype :

```
QuadTree *charger(const char *fichier);
```

Comme dans la question précédente, vous ne gérerez par les erreurs éventuelles à l'ouverture du fichier.

Solution:

```
QuadTree *charger(const char *fichier) {
  QuadTree *qt=NULL;
  FILE *f=fopen(fichier,"r");
  char buffer[200];
  int id, etat;
  double x,y;
  while(fgets(buffer,200, f)) {
    sscanf(buffer,"%d␣%d␣%lf␣%lf",&id,&etat,&x,&y);
    Velo *v=malloc(sizeof(Velo));
    v->id=id;
    v->etat=etat;
    v->x=x;
    v->y=y;
    qt=inserer(qt,v);
  }
  fclose(f);
  return qt;
}
```

Question 5 (3 points)

Écrivez la fonction de libération de la mémoire associée à un QuadTree. Vous prendrez soin de libérer ici la mémoire associée au vélo.

Prototype :

```
void liberer_qt(QuadTree *qt);
```

Solution:

```
void liberer_qt(QuadTree *qt) {
  if (qt) {
    liberer_qt(qt->NW);
    liberer_qt(qt->NE);
    liberer_qt(qt->SW);
    liberer_qt(qt->SE);
    free(qt->velo);
    free(qt);
  }
}
```

2 Liste de vélos

Le résultat d'une recherche dans le QuadTree sera renvoyé sous la forme d'une liste chaînée de vélos. La structure correspondante est :

```
typedef struct _elt {
  Velo *velo;
  struct _elt * suivant;
} Elt;
```

Question 6 (6 points)

Écrivez une fonction permettant de concaténer deux listes. Prototype :

```
Elt *concatener(Elt *liste1, Elt *liste2);
```

La liste `liste2` sera ajoutée à la fin de `liste1`. Il n'est pas nécessaire de copier les éléments.

Solution:

```
Elt *concatener(Elt *liste1, Elt *liste2) {
  Elt *tmp=liste1;
  if (liste1==NULL) {
    return liste2;
  }
  while(tmp->suivant) {tmp=tmp->suivant;}
  tmp->suivant=liste2;
  return liste1;
}
```

Question 7 (3 points)

Écrivez une fonction permettant de libérer la mémoire associée à une liste (les vélos ne seront pas libérés ici).

Prototype :

```
void detruire(Elt *liste);
```

Solution:

```
void detruire(Elt *liste) {
  while(liste) {
    Elt *tmp=liste->suivant;
    free(liste);
    liste=tmp;
  }
}
```

3 Utilisation du QuadTree

Question 8 (9 points)

Dans cette question, vous allez écrire la fonction de recherche que des utilisateurs ou des réparateurs vont utiliser. Il s'agit donc d'utiliser la structure `QuadTree` pour rechercher les vélos dans un état spécifié en argument (les utilisateurs voudront un vélo en bon état et les réparateurs s'intéresseront aux vélos endommagés) et qui sont dans un certain périmètre. La fonction prend donc en argument l'arbre dans lequel les vélos sont rangés, la position (x,y) autour de laquelle chercher, le rayon de la recherche et l'état des vélos. Le résultat est une liste chaînée des vélos qui sont dans l'état demandé et à une distance du point de recherche inférieure au rayon.

La recherche va se dérouler de la façon suivante :

— Si la distance entre la position du point de recherche et celle du vélo de la racine est inférieure au rayon :
  — si l'état est celui attendu, le vélo est ajouté à la liste des résultats
  — chaque secteur adjacent peut contenir des vélos, la recherche continue donc dans les 4 secteurs.
— Si l'écart entre les abscisses est inférieur au rayon, la recherche doit continuer dans les secteurs à l'ouest et à l'est. Sinon, elle ne va continuer qu'à l'ouest ou à l'est selon la position du point de recherche par rapport à celle du vélo de la racine.
— Si l'écart entre les ordonnées est inférieur au rayon, la recherche doit continuer dans les secteurs au nord et au sud. Sinon, elle ne va continuer qu'au nord ou au sud selon la position du point de recherche par rapport à celle du vélo de la racine.
— Le résultat de la recherche est la concaténation des listes obtenues dans les recherches récursives avec, éventuellement, le vélo de la racine du nœud courant.

Vous pourrez utiliser la fonction distance suivante :

```
double distance(double x1, double y1, double x2, double y2) {
  return sqrt(pow(x1-x2,2)+pow(y1-y2,2));
}
```

Il est rappelé que la fonction `fabs` permet de calculer la valeur absolue d'un nombre réel.

Prototype :

```
Elt *chercher_velos(QuadTree *qt, double x, double y, double rayon,
   Etat etat);
```

Solution:

```
Elt *chercher_velos(QuadTree *qt, double x, double y, double rayon,
   Etat etat) {
```

```
if(qt==NULL) {
  return NULL;
}
Elt *res=NULL;
Elt *nelt=NULL;
int secteur[4]={1,1,1,1}; // NW, NE, SE, SW
//  printf("Chercher x=%lf, y=%lf, etat=%d qt x=%lf y=%lf, rayon=%
   lf etat=%d\n",x,y,etat, qt->velo->x,qt->velo->y, rayon, qt->
   velo->etat);
// printf("Fils %p %p %p %p\n",qt->NW, qt->NE, qt->SE, qt->SW);
if(distance(qt->velo->x,qt->velo->y,x,y)<rayon) {
  //printf("=== Velo dans le rayon !! ===\n");
  if (qt->velo->etat==etat) {
    nelt=malloc(sizeof(Elt));
    nelt->velo=qt->velo;
    nelt->suivant=res;
    res=nelt;
    //printf("+++ Ajoute ! +++\n");
  }
}
else {
  if(fabs(qt->velo->x-x)>rayon) {
    if(x<qt->velo->x) {
      secteur[1]=0;
      secteur[2]=0;
    } else {
      secteur[0]=0;
      secteur[3]=0;
    }
  }
  if(fabs(qt->velo->y-y)>rayon) {
    if(y<qt->velo->y) {
      secteur[0]=0;
      secteur[1]=0;
    } else {
      secteur[2]=0;
      secteur[3]=0;
    }
  }
}
//printf("secteurs: %d %d %d %d\n",secteur[0], secteur[1], secteur
   [2], secteur[3]);
if (secteur[0]) {
  nelt=chercher_velos(qt->NW, x, y, rayon, etat);
  res=concatener(res, nelt);
}
if (secteur[1]) {
```

```
    nelt=chercher_velos(qt->NE, x, y, rayon, etat);
    res=concatener(res, nelt);
  }
  if (secteur[2]) {
    nelt=chercher_velos(qt->SE, x, y, rayon, etat);
    res=concatener(res, nelt);
  }
  if (secteur[3]) {
    nelt=chercher_velos(qt->SW, x, y, rayon, etat);
    res=concatener(res, nelt);
  }
  return res;
}
```

Question 9 (9 points)

Les réparateurs vont aller d'un point à un autre dans un certain rayon correspondant au déplacement maximum qu'ils peuvent réaliser en une étape. Lorsqu'ils ont atteint un point de destination, ils garent leur véhicule d'intervention et réparent les vélos qui sont dans une zone située à une distance inférieure ou égale à leur rayon d'action.

Vous allez écrire une fonction qui, en s'appuyant sur le QuadTree, permettra de déterminer où ils doivent aller garer leur véhicule pour le prochain point de leur tournée. Pour simplifier, seules les positions de vélos endommagées seront considérées. Le réparateur sera donc envoyé vers le vélo endommagé qui est entouré du plus grand nombre de vélos également endommagés. Vous allez donc chercher les vélos endommagés autour de la position de départ et dans un rayon correspondant au déplacement maximum du réparateur. Le réparateur sera envoyé à la position du vélo endommagé qui est entouré du plus grand nombre de vélos endommagés.

La fonction renverra la liste des vélos endommagés qui sont dans le périmètre de la position trouvée et permettra de récupérer les coordonnées du point de destination.

Vous n'oublierez pas de libérer la mémoire allouée qui ne sera plus nécessaire après l'exécution de cette fonction.

Prototype :

```
Elt* prochain_point_tournee(QuadTree *qt, double start_x, double
   start_y, double *next_x, double *next_y, double deplacement_max,
   double rayon_action);
```

Les paramètres `next_x` et `next_y` permettront de récupérer les coordonnées du prochain point de la tournée. `deplacement_max` indique le déplacement maximal depuis le point de départ, dont les coordonnées sont `start_x` et `start_y`. `rayon_action` indique le rayon de la zone d'action du réparateur autour de son point de destination. Enfin `qt` contient l'arbre dans lequel chercher les vélos.

Solution:

```
Elt *prochain_point_tournee(QuadTree *qt, double start_x, double
   start_y, double *next_x, double *next_y, double deplacement_max,
   double rayon_action) {
  Elt *candidats = chercher_velos(qt, start_x, start_y,
     deplacement_max, Endommage);
  Elt *tmp=candidats;
```

```
  Elt *res=NULL;
  int nb=0;
  *next_x=start_x;
  *next_y=start_y;

  while(tmp) {
    int nnb=0;
    Elt *candidats_local = chercher_velos(qt, tmp->velo->x, tmp->
       velo->y, rayon_action, Endommage);
    Elt *tmp2=candidats_local;
    while (tmp2) {nnb++;tmp2=tmp2->suivant;}
    printf("point: %lf, %lf nnb:%d\n",tmp->velo->x, tmp->velo->y,
       nnb);
    if (nnb>nb) {
      *next_x=tmp->velo->x;
      *next_y=tmp->velo->y;
      nb=nnb;
      detruire(res);
      res=candidats_local;
    }
    else {
      detruire(candidats_local);
    }
    tmp=tmp->suivant;
  }
  detruire(candidats);
  return res;
}
```

Question 10 (6 points)

Écrivez une fonction `main` permettant de tester vos fonctions de la façon indiquée ci-après. Votre programme prendra le nom du fichier contenant les données des vélos en argument. La position du point de départ, le déplacement maximum et le rayon d'action seront saisis au clavier. Le programme lira le fichier, cherchera le prochain point de tournée et fera passer les vélos réparés de l'état `Endommage` à l'état `Usage` avant d'écrire les données ainsi modifiées dans le fichier. Vous prendrez soin de libérer la mémoire allouée.

Solution:

```
int main(int argc, char **argv) {
  if (argc!=2) {
    printf("Usage: %s nom_de_fichier\n", argv[0]);
    return 1;
  }
  QuadTree *qt=charger(argv[1]);
  afficher(qt);
  double start_x,start_y, next_x, next_y,dmax,raction;
  printf("Saisir la position de depart: x y\n");
  scanf(" %lf %lf",&start_x,&start_y);
```

```
  printf("Saisir␣le␣deplacement␣max␣et␣le␣rayon␣d'action\n");
  scanf("␣%lf␣%lf",&dmax,&raction);
  Elt *velos=prochain_point_tournee(qt,start_x,start_y,&next_x,&
     next_y,dmax,raction);
  printf("Prochain␣point␣de␣tournee:␣%lf␣%lf\n",next_x, next_y);
  Elt *tmp=velos;
  while(tmp) {
    tmp->velo->etat=Usage;
    tmp=tmp->suivant;
  }
  sauvegarder(qt,"qd2.txt");
  detruire(velos);
  liberer_qt(qt);
  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.

```
typedef enum _etat {Neuf=0,Usage, Endommage} Etat;

typedef struct _velo {
  int id;
  Etat etat;
  double x;
  double y;
} Velo;

typedef struct _elt {
  Velo *velo;
  struct _elt * suivant;
} Elt;

Elt *concatener(Elt *liste1, Elt *liste2);
void detruire(Elt *liste);

typedef struct _quadtree QuadTree;
struct _quadtree {
  Velo *velo;
  QuadTree *NW;
  QuadTree *NE;
  QuadTree *SW;
  QuadTree *SE;
};

QuadTree *creer_quadtree(Velo *v);

QuadTree *inserer(QuadTree *qt, Velo *v);

Elt *chercher_velos(QuadTree *qt, double x, double y, double rayon,
   Etat etat);

Elt* prochain_point_tournee(QuadTree *qt, double start_x, double
   start_y, double *next_x, double *next_y, double deplacement_max,
   double rayon_action);

void ecrire_quadtree(QuadTree *qt, FILE *f);
void sauvegarder(QuadTree *qt, const char *fichier);

QuadTree *charger(const char *fichier);

void afficher(QuadTree *qt);
```