SCIENCES SORBONNE UNIVERSITÉ

Votre numéro d'anonymat : | | | |

Programmation et structures de données en C– LU2IN018

Examen du 21 janvier 2021
1 heure 30
Aucun document n'est autorisé.

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

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

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

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

Collection de Minéraux

Sorbonne Université dispose d'une magnifique collection de minéraux qui est aussi l'une des plus anciennes de France (n'hésitez pas à aller la voir lorsque ce sera de nouveau possible, l'accès est à côté de la tour 25).

Vous allez écrire du code pour gérer une collection de ce type et faire des recherches à partir de la composition des minéraux qu'elle contient.

1 Gestion des minéraux

Les minéraux peuvent être décrits par leur dureté et leur densité, qui sont des nombres réels, ainsi que par leur éclat, qui est une chaîne de caractères. Ils ont également un nom et une composition chimique. Ils vont être manipulés dans votre programme au travers de la structure suivante :

```
typedef struct _Mineral {
  char *nom;
  L_element *composition;
  float durete;
  float densite;
  char *eclat;
} Mineral;
```

La composition chimique est une liste chaînée des éléments qui constituent le minéral. On ne considèrera pas ici la formule chimique exacte, mais juste les atomes qui composent le minéral. La Goethite, par exemple, a pour formule $Fe^{3+}OOH$. Nous considèrerons donc que sa composition est : Fe, O et H. Cette composition sera représentée par la liste chaînée des numéros atomiques des éléments qui composent le minéral (1 pour H, 6 pour O, 26 pour Fe, ...). La composition s'appuiera donc sur la structure suivante :

```
typedef struct _L_element {
  int numero;
  struct _L_element *suivant;
} L_element;
```

La correspondance entre un numéro atomique et le symbole chimique associé se fera au travers d'un tableau `tableau_elements` déclaré comme une variable globale :

```
#define NB_ELEM 118
char *tableau_elements[NB_ELEM];
```

Soit les instructions suivantes, placées dans le main :

```
int num=6;
printf("Symbole␣associé␣au␣numéro␣atomique␣%d:␣%s\n", num,
   tableau_elements[num-1]);
```

Leur exécution affichera :

```
Symbole associé au numéro atomique 6: C
```

Le carbone (C) est en effet l'élément chimique qui a pour numéro atomique 6.

Remarquez que le tableau commençant à 0 et non à 1, il faut retirer 1 au numéro atomique pour être dans la bonne case.

Question 1 (3 points)

Écrivez une fonction permettant d'afficher la composition chimique. Cette fonction parcourt la liste chaînée et affiche les symboles associés aux numéros atomiques qui la composent. Prototype :

```
void affiche_comp(L_element *pcomp);
```

Exemple d'affichage :

```
H Fe O
```

Solution:

```
void affiche_comp(L_element *pcomp) {
  while (pcomp!=NULL) {
    printf("%s ", tableau_elements[pcomp->numero-1]);
    pcomp = pcomp->suivant;
  }
}
```

Question 2 (6 points)

Écrivez une fonction permettant de chercher si un élément fait partie d'une composition à partir de son symbole. Prototype :

```
int cherche_comp(L_element *pcomp, char *el);
```

La fonction renverra 1 si l'élément est trouvé et 0 sinon.

La fonction doit commencer par trouver le numéro atomique à partir de la chaîne de caractère de son symbole. Pour cela, elle doit parcourir le tableau à la recherche du bon symbole avant de parcourir la liste chaînée de la composition à la recherche de ce numéro.

Solution:

```
int cherche_comp_num(L_element *pcomp, int num_el) {
  while(pcomp) {
    if (pcomp->numero == num_el)
      return 1;
    pcomp = pcomp->suivant;
  }
  return 0;
}

int cherche_comp(L_element *pcomp, char *el) {
  //recherche du numéro à partir de la chaîne de caractère (un
     dictionnaire aurait été très bien ici...)
  int i;
  for (i=0;i<NB_ELEM; i++) {
    if (strcmp(el,tableau_elements[i])==0) {
      return cherche_comp_num(pcomp, i+1);
    }
  }
  return 0; // symbole inconnu
}
```

La solution s'appuie sur deux fonctions, mais ce n'est pas du tout indispensable.

Question 3 (6 points)

Écrivez une fonction permettant d'allouer un minéral. Prototype :

```
Mineral *allouer_min(char *nom, L_element *pcomp, float durete, float
    densite, char *eclat);
```

Les chaînes de caractères (`nom` et `eclat`) transmises en argument à la fonction sont susceptibles d'êtres libérées juste après l'appel. Par contre, la composition sera supposée pérenne et n'aura pas besoin d'être recopiée.

Solution:

```
Mineral *allouer_min(char *nom, L_element *pcomp, float durete,
   float densite, char *eclat) {
  Mineral *nm = (Mineral *) malloc (sizeof(Mineral));
  nm->nom = strdup(nom);
  nm->composition=pcomp;
  nm->durete = durete;
  nm->densite = densite;
  nm->eclat = strdup(eclat);
  return nm;
}
```

Question 4 (2 points)

Écrivez une fonction permettant de libérer la mémoire associée à un minéral. Prototype :

```
void liberer_min(Mineral *pmin);
```

Vous prendrez soin de libérer toute la mémoire occupée par le minéral, composition incluse.

Solution:

```
void liberer_comp(L_element *pcomp) {
  while(pcomp) {
    L_element *tmp=pcomp->suivant;
    free(pcomp);
    pcomp=tmp;
  }
}

void liberer_min(Mineral *pmin) {
  free(pmin->nom);
  liberer_comp(pmin->composition);
  free(pmin->eclat);
  free(pmin);
}
```

La solution passe par une fonction dédiée à la libération de la mémoire associée à la composition, mais cela n'a rien d'obligatoire, La boucle de libération de la composition peut être incluse directement dans la fonction de libération d'un minéral.

2 Gestion de collections

Une collection de minéraux est un ensemble d'échantillons. Un échantillon associe à un minéral une provenance et un emplacement (le numéro de la vitrine dans laquelle il est exposé). Les échantillons seront représentés avec la structure suivante :

```
typedef struct _Echantillon {
  Mineral *mineral;
  char *provenance;
  int emplacement;
} Echantillon;
```

La collection sera une liste chaînée d'échantillons :

```
typedef struct _Collection {
  Echantillon *echantillon;
  struct _Collection *suivant;
} Collection;
```

Question 5 (3 points)

Écrivez une fonction d'ajout d'un échantillon à une collection. Prototype :

```
Collection *ajout_coll(Collection *pcoll, Echantillon *pech);
```

La fonction renvoie la collection résultant de l'ajout de l'échantillon à la collection transmise en argument. L'échantillon ne doit pas être dupliqué. La collection n'est pas triée, vous pouvez donc faire une insertion en tête.

Solution:

```
Collection *ajout_coll(Collection *pcoll, Echantillon *pech) {
  Collection *nc =(Collection *)malloc(sizeof(Collection));
  nc->echantillon = pech;
  nc->suivant=pcoll;
  return nc;
}
```

Question 6 (6 points)

Écrivez une fonction de suppression d'un échantillon d'une collection. Prototype :

```
Collection *supprimer_coll(Collection *pcoll, Echantillon *pech);
```

La fonction renvoie la collection modifiée. Vous devrez gérer le cas d'une collection vide et le cas où l'échantillon est au début de la collection. Pensez à libérer l'élément `Collection` qui contient l'échantillon. Par contre, il ne faut pas libérer l'échantillon.

Solution:

```
Collection *supprimer_coll(Collection *pcoll, Echantillon *pech) {
  if (pcoll == NULL) return NULL;
  if (pech == pcoll->echantillon) {
    Collection *nc = pcoll->suivant;
    free(pcoll);
    return nc;
  }
  Collection *debut = pcoll;
  while (pcoll->suivant!=NULL) {
    if (pech == pcoll->suivant->echantillon) {
      Collection *nc = pcoll->suivant;
      pcoll->suivant = nc->suivant;
```

```
      free(nc);
      return debut;
    }
    pcoll=pcoll->suivant;
  }
  return debut;
}
```

3 Lecture d'une collection

Exemple de contenu d'un fichier contenant 2 échantillons (Goethite, lignes 1 à 7 et Fluorite, lignes 8 à 14) :

```
1  Goethite
2  26 8 1
3  5
4  4.3
5  métallique à mat
6  amérique
7  24
8  Fluorite
9  20 9
10 4
11 3.1
12 vitreux
13 europe
14 12
```

Le format est le suivant :

```
nom de minéral
composition
dureté
densité
éclat
provenance
numéro de vitrine
```

Les 5 premières lignes correspondent au minéral. L'échantillon est décrit par son minéral, sa provenance et son numéro de vitrine.

Les différentes fonctions de lecture renverront la valeur NULL en cas d'échec (par exemple si la fin du fichier est atteinte). On fera l'hypothèse que le fichier est correct, c'est à dire que si le minéral a pu être lu, sa provenance et son emplacement seront présents dans le fichier sans avoir besoin de le vérifier.

Dans la suite, vous pourrez utiliser les fonctions de lecture d'un minéral et d'allocation d'un échantillon QUE VOUS N'AUREZ PAS À ÉCRIRE. Prototypes :

```
Mineral *lire_min(FILE *f);
```

```
Echantillon *allouer_ech(Mineral *mineral, char *provenance, int
   emplacement);
```

Question 7 (3 points)

Écrivez la fonction de lecture d'un échantillon qui consiste à lire le minéral puis la provenance et l'emplacement. Prototype :

```
Echantillon *lire_ech(FILE *f);
```

Elle prend en argument le pointeur sur le fichier déjà ouvert et renvoie l'échantillon lu.

Solution:

```
Echantillon *lire_ech(FILE *f) {
  char provenance[100];
  int emplacement;
  Mineral *mineral = lire_min(f);
  if (mineral==NULL) return NULL;
  fgets(provenance, 100, f);
  provenance[strlen(provenance)-1]='\0';
  fscanf(f,"%d\n", &emplacement);
  return allouer_ech(mineral, provenance, emplacement);
}
```

Question 8 (3 points)

Écrivez la fonction de lecture d'une collection. La fonction de lecture d'une collection devra ouvrir le fichier dont le nom lui a été transmis et faire une boucle pour lire tous les échantillons de la collection avant de fermer le fichier ouvert et de renvoyer la collection lue. Prototype :

```
Collection *lire_coll(const char *nom_fichier);
```

Solution:

```
Collection *lire_coll(const char *nom_fichier) {
  FILE *f=fopen(nom_fichier, "r");
  Collection *c=NULL;
  Echantillon *ne=lire_ech(f);
  while(ne) {
    c=ajout_coll(c,ne);
    ne=lire_ech(f);
  }
  fclose(f);
  return c;
}
```

4 Choix des minéraux

Nous allons maintenant écrire des fonctions permettant de diriger les visiteurs vers les vitrines qui les intéressent. Pour cela, vous allez écrire une fonction permettant de poser des questions sur les échantillons que les visiteurs souhaitent voir, ces questions porteront sur la composition des minéraux. Vous allez pour cela utiliser un arbre binaire de décision.

Dans un tel arbre, chaque noeud interne de l'arbre contient un numéro atomique. Ces noeuds ont deux sous-arbres : un sous-arbre `sans` qui permet de poursuivre la recherche avec les échantillons qui ne contiennent pas cet élément et un sous-arbre `avec` qui permet de poursuivre avec les échantillons qui le contiennent.

Chaque noeud (interne ou feuille) contient une collection, qui est la collection des échantillons qui respecte les choix indiqués jusqu'à présent : à la racine, c'est l'ensemble de la collection et sur les autres noeuds, ce sont les échantillons qui correspondent aux choix effectués lors des branchements précédents.

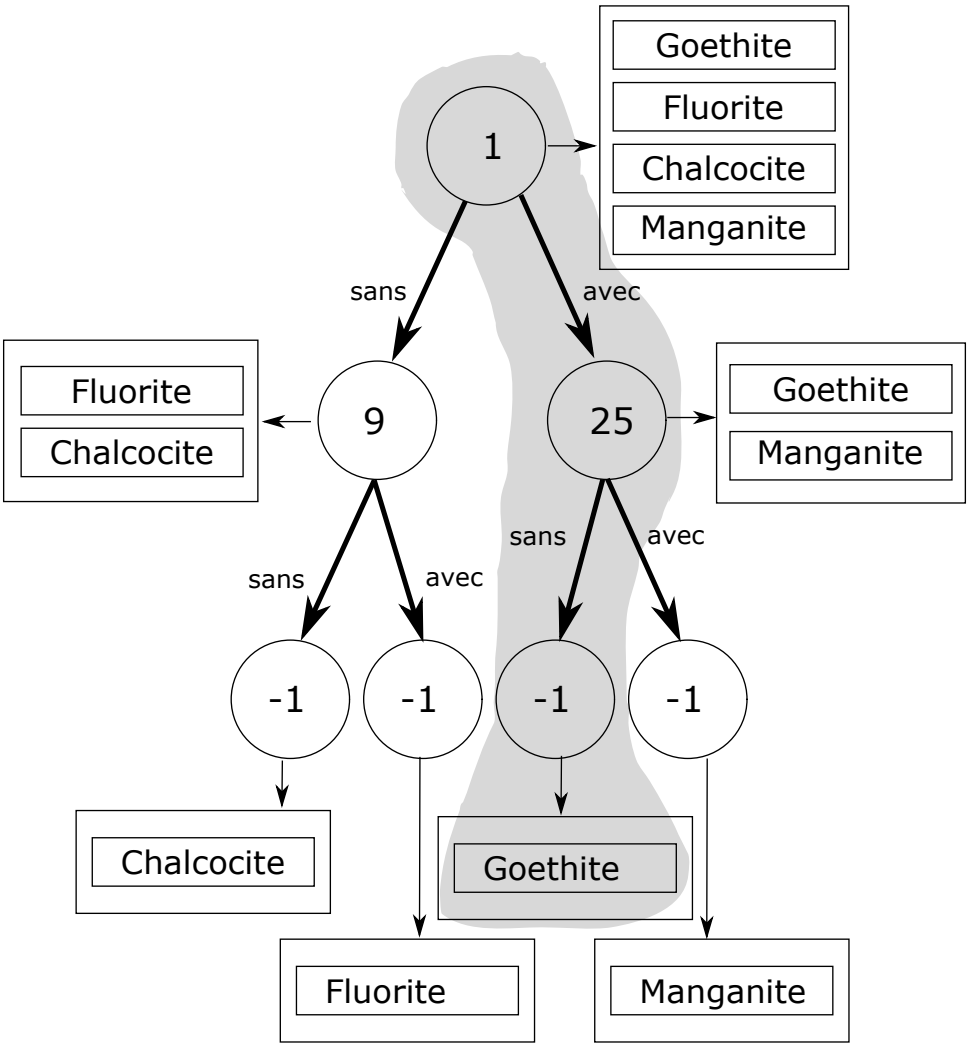

FIGURE 1 – Exemple d'arbre de décision. Les noeuds sont représentés par des cercles. Les noeuds internes contiennent les numéros atomiques des éléments qui vont permettre de réduire la collection selon qu'il sera ou pas dans un échantillon donné. Les feuilles n'ont pas d'élément associé (d'où la valeur `-1`). La collection considérée contient 4 échantillons représentés sur la figure par le nom du minéral uniquement pour simplifier. Dans cette collection, il n'y a donc qu'un échantillon de Goethite qui contient de l'hydrogène (H, de numéro atomique 1) et pas de manganèse (Mn, numéro atomique 25) dans sa composition.

La figure 1 représente un exemple d'un tel arbre. Un visiteur indiquant qu'il souhaite voir un échantillon contenant de l'hydrogène et pas de manganèse sera orienté vers l'échantillon de Goethite de la collection.

La structure d'arbre utilisée est la suivante :

```
typedef struct _Arbre_decision {
```

```
  int num_el;
  struct _Arbre_decision *sans;
  struct _Arbre_decision *avec;
  Collection *coll;
} Arbre_decision;
```

Question 9 (6 points)

Écrivez la fonction d'utilisation de l'arbre. Prototype :

```
void trouver_ech(Arbre_decision *pad);
```

Cette fonction utilisera une boucle dans laquelle, à chaque itération, sera demandé au visiteur s'il souhaite voir des échantillons contenant l'élément dont le numéro atomique est contenu dans le noeud. S'il répond 'o' (tapé au clavier), le parcours continue avec le sous-arbre `avec`, sinon, il continue avec le sous-arbre `sans`. Le parcours s'arrête lorsqu'une feuille est atteinte ou lorsque le visiteur indique qu'il souhaite arrêter (la question lui sera posée à chaque itération).

Exemple d'interaction :

```
Voulez-vous que les échantillons contiennent du H ? (o/n)o
Voulez-vous continuer le parcours ? (o/n)o
Voulez-vous que les échantillons contiennent du Mn ? (o/n)n
Echantillons trouvés:
Goethite, vitrine: 24
```

Vous pourrez utiliser la fonction d'affichage de collection de prototype suivant (pas besoin de la définir) :

```
void affiche_coll(Collection *pcoll);
```

Solution:

```
void trouver_ech(Arbre_decision *pad) {
  int cont=1;
  while(cont) {
    printf("\nVoulez-vous␣que␣les␣échantillons␣contiennent␣du␣%s␣?␣(
       o/n)", tableau_elements[pad->num_el-1]);
    char reponse;
    scanf("␣%c",&reponse);
    if(reponse=='o')
      pad=pad->avec;
    else
      pad=pad->sans;
    if (pad->num_el==-1)
      cont=0;
    else {
      printf("␣␣Voulez-vous␣continuer␣le␣parcours␣?␣(o/n)");
      scanf("␣%c",&reponse);
      if (reponse == 'n')
        cont=0;
    }
  }
```

```
  printf("Echantillons␣trouvés:␣\n");
  affiche_coll(pad->coll);
  printf("\n");
}
```

Question 10 (6 points)

Écrivez un main permettant de lire la collection contenue dans le fichier "coll.txt", de construire l'arbre de décision et de demander à l'utilisateur ce qu'il souhaite voir. Vous prendrez soin de libérer toute la mémoire allouée. Vous pourrez utiliser pour cela les fonctions dont le prototype est indiqué à la fin de l'énoncé sans avoir besoin de les définir. En particulier, la construction de l'arbre se fera avec un appel à la fonction suivante :

```
Arbre_decision *construire_arbre(Collection *pcoll, L_element *
   a_ignorer);
```

Le deuxième argument est utilisé pour la récursion sur laquelle s'appuie cette fonction. Vous pourrez mettre ici la valeur `NULL`.

Les prototypes des fonctions sont dans le fichier "mineraux.h".

Solution:

```
#include "mineraux.h"


int main(void) {
  Collection *pcoll = lire_coll("coll.txt");

  Arbre_decision *pad=construire_arbre(pcoll, NULL);
  trouver_ech(pad);

  liberer_arbre(pad);
  liberer_coll(pcoll, 1);
  return 0;
}
```

Question 11 (6 points)

Lors de l'exécution du programme, une erreur de segmentation a été observée. Le main a été modifié pour juste créer l'arbre, l'afficher et l'effacer. Le problème continue de se produire.

Le programme a donc été relancé avec valgrind. Cela a fait apparaître les points suivants :

```
$ valgrind ./main_mineraux_bug

[...]

==31470== Conditional jump or move depends on uninitialised value(s)
==31470==    at 0x10A8D9: afficher_arbre (mineraux_bug.c:448)
==31470==    by 0x10A983: afficher_arbre (mineraux_bug.c:458)
==31470==    by 0x109662: main (main_mineraux_bug.c:8)

[...]
```

```
==31470== Conditional jump or move depends on uninitialised value(s)
==31470==    at 0x10A882: liberer_arbre (mineraux_bug.c:439)
==31470==    by 0x10A8A3: liberer_arbre (mineraux_bug.c:441)
==31470==    by 0x10966E: main (main_mineraux_bug.c:10)

[...]
```

Les lignes incriminées sont les suivantes :

```
438   if(pad) {
439     liberer_arbre(pad->avec);
440     liberer_arbre(pad->sans);
441     liberer_coll(pad->coll, 0);
442     free(pad);
443   }
444 }
```

```
447   if(pad) {
448     if (pad->num_el==-1) {
449       printf("< ");
450       affiche_coll(pad->coll);
451       printf(" > ");
452     }
453     else {
454       printf(" ( %s <", tableau_elements[pad->num_el-1]);
455       affiche_coll(pad->coll);
456       printf("> sans: ");
457       afficher_arbre(pad->sans);
458       printf(" avec: ");
459       afficher_arbre(pad->avec);
460       printf(") ");
461     }
462   }
463
464 }
```

On vous suggère que le problème vient de la fonction de construction de l'arbre réalisée dans la fonction :

```
Arbre_decision *construire_arbre(Collection *pcoll, L_element *
   a_ignorer);
```

Cette fonction détermine selon quel élément scinder la collection (son numéro atomique est `i_ref+1`) et construit les collections résultant de la scission (`pcoll_avec` et `pcoll_sans`).

La construction de l'arbre est alors réalisée avec les instructions suivantes :

```
415   if ((taille_coll(pcoll_avec)==0)||(taille_coll(pcoll_sans)==0)) {
416     printf("Feuille atteinte...\n");
```

```
417       na->num_el=-1;
418       na->coll = duplique_coll(pcoll);
419       na->avec=NULL;
420       na->sans=NULL;
421     }
422     else {
423       printf("Numero␣de␣l'élément␣à␣ajouter␣à␣ce␣niveau:␣%d\n",i_ref+1)
            ;
424       na->num_el=i_ref+1;
425       na->coll=duplique_coll(pcoll);
426       construire_arbre(pcoll_avec, ne);
427       construire_arbre(pcoll_sans, ne);
428     }
429     liberer_coll(pcoll_avec, 0);
430     liberer_coll(pcoll_sans, 0);
431     free(ne);
432     return na;
433   }
```

Qu'est-ce qui permet de faire cette déduction sur l'emplacement de la ou des erreurs ? Où se situe le problème (indiquer le ou les numéros de ligne et l'erreur ou les erreurs qu'elles contiennent) ?

Solution: Valgrind indique que des valeurs n'ont pas été initialisées. Ces valeurs correspondent à des pointeurs sur des noeuds de l'arbre. L'arbre n'a pas été manipulé, il est donc tel qu'il a été créé, aussi si des pointeurs non initialisés ont été trouvés, ils résultent de la création de l'arbre.

En observant le code de création de l'arbre, on peut voir que le résultat des appels à `construire_arbre` n'est pas récupéré (lignes 427 et 428). Les sous-arbres créés ne sont donc pas insérés à leur place et les champs `sans` et `avec` des noeuds ne sont donc pas initialisés, d'où le message d'erreur de valgrind.

Question 12 (3 points)

Question bonus

Les compositions et les collections s'appuient toutes les deux sur des listes chaînées. Nous aurions donc pu utiliser une biliothèque générique de liste. Est-ce que les fonctions `affiche_comp`, `cherche_comp`, `ajout_coll`, peuvent se remplacer facilement par leur version générique ? Si oui est-ce qu'une fonction spécifique à la donnée portée par la liste est nécessaire ? Dans ce cas, préciser laquelle.

Solution: Pour `affiche_comp`, il n'y a aucun problème à utiliser une version générique. Il faut alors fournir une fonction permettant d'afficher une composition.

Pour `cherche_comp`, c'est plus délicat car cette fonction cherche la composition à partir du symbole et non de l'entier qui est contenu dans un élément de la liste. On peut garder cette fonction pour trouver le bon numéro et ensuite faire appel à la fonction générique de recherche de l'entier dans la liste. Dans ce cas, on aura besoin d'une fonction de comparaison d'entiers.

`ajout_coll` fait un ajout en tête. Cela ne pose aucun problème et ne nécessite pas de manipuler les données car la version demandée ne duplique pas l'échantillon.

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.

```
#ifndef _MINERAUX_H_
#define _MINERAUX_H_

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define NB_ELEM 118
extern char *tableau_elements[NB_ELEM];



typedef struct _L_element {
  int numero;
  struct _L_element *suivant;
} L_element;

/* Affichage d'une composition (Q1) */
void affiche_comp(L_element *pcomp);

/* Recherche d'un élément à partir de son symbole (
   Q2) */
int cherche_comp(L_element *pcomp, char *el);

typedef struct _Mineral {
  char *nom;
  L_element *composition;
  float durete;
  float densite;
  char *eclat;
} Mineral;

/* Allocation d'un minéral (Q3) */
Mineral *allouer_min(char *nom, L_element *pcomp,
   float durete, float densite, char *eclat);

/* Libération d'un minéral (Q4) */
void liberer_min(Mineral *pmin);

typedef struct _Echantillon {
  Mineral *mineral;
  char *provenance;
  int emplacement;
} Echantillon;

typedef struct _Collection {
  Echantillon *echantillon;
  struct _Collection *suivant;
} Collection;

/* Ajout dans une collection (Q5) */
Collection *ajout_coll(Collection *pcoll,
   Echantillon *pech);

/* Suppression d'une collection (Q6) */
Collection *supprimer_coll(Collection *pcoll,
   Echantillon *pech);

/* Lire un minéral (utile pour Q7) */
Mineral *lire_min(FILE *f);

/* Allouer un échantillon (utile pour Q7) */
Echantillon *allouer_ech(Mineral *mineral, char *
   provenance, int emplacement);

/* Lire un échantillon (Q7) */
Echantillon *lire_ech(FILE *f);

/* Lire une collection (Q7) */
Collection *lire_coll(const char *nom_fichier);

/* Renvoie la copie d'une collection transmise en
```

```
   argument */
Collection *duplique_coll(Collection *pcoll);

/* Libérer une collection, si libere_ech vaut 1, les
    échantillons sont libérés, sinon, ils ne le sont
    pas (utile pour Q9 ?) */
void liberer_coll(Collection *pcoll, int libere_ech)
   ;

/* Libérer un échantillon (utile pour Q9 ?) */
void liberer_ech(Echantillon *pech);

/* Libérer un minéral (Q4) */
void liberer_min(Mineral *pmin);

/* Libérer une composition (utile pour Q9 ?) */
void liberer_comp(L_element *pcomp);

/* Taille d'une collection */
int taille_coll(Collection *pcoll);

/* Affichage d'une collection (utile pour Q8) */
void affiche_coll(Collection *pcoll);

typedef struct _Arbre_decision {
  int num_el;
  struct _Arbre_decision *sans;
  struct _Arbre_decision *avec;
  Collection *coll;
} Arbre_decision;

/* Construction d'un arbre binaire de décision (
   utile pour Q9 ?) */
Arbre_decision *construire_arbre(Collection *pcoll,
   L_element *a_ignorer);

/* Libérer un arbre binaire de décision (utile pour
   Q9 ?) */
void liberer_arbre(Arbre_decision *pad);

/* Affichage d'un arbre */
void afficher_arbre(Arbre_decision *pad);

/* Trouver un échantillon à partir d'un arbre de dé
   cision (Q8) */
void trouver_ech(Arbre_decision *pad);

#endif
```