Nom :	
Prénom :	
No. groupe :	
No. carte :	

Programmation et structures de données en C– LU2IN018

Partiel du 14 novembre 2019

1h30

Aucun document n'est autorisé. Le memento qui a été distribué est reproduit à la fin de cet énoncé.

Les calculatrices, baladeurs et autres appareils électroniques sont interdits. Les téléphones mobiles doivent être éteints et rangés dans les sacs.

Toutes les questions sont indépendantes. Pour les questions à choix multiples à 1 point, vous obtenez 1 point si vous avez coché toutes les cases correspondant à des réponses justes et seulement celles-ci. Pour les questions à 2 points ou plus acceptant plusieurs réponses, vous perdez 1 point par réponse erronée (case juste non cochée ou case fausse cochée). La note minimale à une question est 0. Le barème sur 20 points (15 questions) n'a qu'une valeur indicative.

ATTENTION : lisez le sujet dans son intégralité avant de commencer. Certaines questions sont à réponse libre (lecture ou écriture de code). Elles peuvent donc nécessiter plus de temps de réflexion que les questions à choix multiples.

Il ne vous est pas demandé de vérifier qu'un `malloc` a bien alloué la mémoire. De même il n'est pas demandé de vérifier qu'un `fopen` a ouvert correctement le fichier demandé.

Question 1 (2 points)

Écrire une instruction permettant d'allouer dynamiquement un tableau de 15 entiers dans une variable nommée `tab` (sans faire de test pour vérifier si cela s'est bien passé).

Solution:

```
int *tab = (int *)malloc(15*sizeof(int));
```

Pas obligatoire de mettre le (int *) devant malloc.

Question 2 (1 point)

Comment libérer la mémoire allouée à la question précédente ? Cochez la ou les affirmations correctes :

☐
```
tab = NULL;
```

■
```
free(tab);
```

☐
```
int i;
for (i=0;i<15;i++) {
   free(tab[i]);
}
```

☐ `free(*tab);`

☐ `free(&tab);`

Question 3 (1 point)

On veut écrire une fonction pour multiplier tous les éléments d'un tableau d'entiers précédemment alloués par un entier. Quel est ou quels sont les prototypes appropriés ?

■ `void mult(int tab[], int taille, int coeff);`

☐ `void mult(int *tab[], int taille, int coeff);`

☐ `void mult(int tab, int taille, int coeff);`

☐ `int *mult(int taille, int coeff);`

■ `void mult(int *tab, int taille, int coeff);`

Question 4 (2 points)

Écrivez l'implémentation de la fonction `mult` en choisissant l'un des prototypes précédents (à préciser dans le cas où vous en avez coché plusieurs).

Solution:

```
void mult(int *tab, int taille, int coeff) {
   int i;
   for (i=0;i<taille;i++) {
      tab[i]*=coeff;
   }
}
```

On souhaite écrire une fonction pour lire dans un fichier des points 2D. Le format du fichier est le suivant :

```
2
44.5;24.3
67.2;12.6
```

La première ligne indique le nombre de points à lire et les autres lignes contiennent les points, un par ligne, avec la première coordonnée, puis un ' ;', puis la deuxième coordonnée.

La structure utilisée et la fonction, incomplète, sont indiquées ci-dessous :

```
typedef struct _point{
   float x;
   float y;
}point;

point *lire_points(char *nom_fichier, int *nbpts) {
   FILE *f=/* OUVERTURE DU FICHIER */
   if (f==NULL) {
      fprintf(stderr,"Erreur␣lors␣de␣l'ouverture␣de␣%s\n",nom_fichier
         );
      return NULL;
   }
   int nblignes=0;
   fscanf(f,"␣%d", &nblignes);
   *nbpts=nblignes;
   point *tab=(point *)malloc(sizeof(point)*nblignes);
   if (tab==NULL) {
      fprintf(stderr, "Problème␣d'allocation␣mémoire\n");
      return NULL;
   }
   int i;
   for (i=0;i<nblignes;i++) {
      fscanf(/* LECTURE D'UN POINT */);
   }
   fclose(f);
   return tab;
}
```

Question 5 (1 point)

Par quoi faut-il remplacer

```
FILE *f=/* OUVERTURE DU FICHIER */
```

Cochez la ou les affirmations correctes :

- ■ `FILE *f=fopen(nom_fichier, "r");`
- ☐ `FILE *f=fopen(nom_fichier, "w");`
- ☐ `FILE *f=fopen("nom_fichier", "r");`
- ☐ `FILE *f=fopen("nom_fichier", "w");`

Question 6 (1 point)

Par quoi faut-il remplacer

```
fscanf(/* LECTURE D'UN POINT */);
```

Cochez la ou les affirmations correctes :

- ■ `fscanf(f,"␣%f␣;␣%f",&tab[i].x, &tab[i].y);`

☐ `fscanf(f,"␣%f␣;␣%f",*tab[i].x, *tab[i].y);`

☐ `fscanf(f,"␣%f␣;␣%f",&(tab+i).x, &(tab+i).y);`

■ `fscanf(f,"␣%f␣;␣%f",&(tab+i)->x, &(tab+i)->y);`

☐ `fscanf(f,"␣%f␣;␣%f",tab[i].x, tab[i].y);`

☐ `fscanf(f,"␣%f␣;␣%f",tab[i]->x, tab[i]->y);`

Question 7 (3 points)

Écrivez une fonction main pour tester la fonction précédente. Votre fonction lira le tableau contenu dans le fichier `mes_donnees.txt` en faisant appel à la fonction précédente, affichera le contenu du fichier ainsi lu et libèrera la mémoire qui a été allouée avant de quitter le programme.

Affichage du contenu du fichier précédent :

```
point[0]: x=44.5 y=24.3
point[1]: x=67.2 y=12.6
```

Solution:

```
int main(void) {
   int nbpts=0;
   point *tab=lire_points("mes_donnees.txt",&nbpts);
   int i;
   for (i=0;i<nbpts;i++) {
      printf("point[%d]:␣x=%f␣y=%f\n",i,tab[i].x, tab[i].y);
   }
   free(tab);
   return 0;
}
```

La fonction `lire_points` a été mise dans le fichier `points.c`. Le prototype de la fonction a été écrit dans `points.h`. La fonction `main` a été mise dans le fichier `mon_main.c`.

Question 8 (1 point)

Que faut-il ajouter au début du fichier `mon_main.c` pour pouvoir utiliser la fonction `lire_points` ?

Solution:

```
#include "points.h"
```

Il est correct également de donner, à la place de cette ligne, le prototype de la fonction, c'est-à-dire :

```
point * lire_points(char *nom_fichier, int *nbpts);
```

Question 9 (1 point)

Comment créer l'exécutable `mon_main` ? Cochez la ou les réponses correctes :

■
```
gcc -Wall -c mon_main.c
gcc -Wall -c points.c
gcc -Wall -o mon_main mon_main.o points.o
```
☐ `gcc -Wall -o mon_main.c points.c`

☐ `gcc -Wall -o mon_main mon_main.c points.h`

■ `gcc -Wall -o mon_main mon_main.c points.c`
□ `gcc -Wall -c mon_main.c`
`gcc -Wall -c points.c`
`gcc -Wall -c points.h`
`gcc -Wall -o mon_main mon_main.o points.o`
□ `gcc -Wall -c mon_main.c`
`gcc -Wall -c points.c`
`gcc -Wall -o points.h mon_main.o points.o`

Question 10 (1 point)

Cochez la ou les réponses correctes :

□ Une liste chaînée de X entiers prend la même place qu'un tableau de X entiers
■ L'accès à un élément d'un tableau est immédiat (sa complexité est indépendante de la taille du tableau)
□ L'accès à un élément d'une liste chaînée est immédiat (sa complexité est indépendante de la taille de la liste)
■ Les éléments d'un tableau alloué dynamiquement sont toujours contigus dans la mémoire disponible
□ Les éléments d'une liste chaînée allouée dynamiquement sont toujours contigus dans la mémoire disponible
■ Une liste de X éléments alloués dynamiquement sera libérée avec X `free`
□ Une implémentation d'un ensemble de données sous forme de liste chaînée est toujours préférable à une implémentation sous forme de tableau

Question 11 (2 points)

Soit une liste chaînée définie avec la structure suivante :

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

Écrivez une fonction permettant de déterminer le nombre d'éléments de la liste. Prototype :

```
int taille_liste(PElement liste);
```

Solution:

```
int taille_liste(PElement liste) {
   int t=0;
   while(liste) {
      t++;
      liste=liste->suivant;
   }
   return t;
}
```

Question 12 (1 point)

La fonction suivante vise à créer un élément ayant un entier en donnée :

```
PElement creer_elt_entier(int data) {
   PElement pelt=(PElement)malloc(sizeof(Element));
```

```
   /* initialisation du champ data */

   pelt->suivant=NULL;
   return pelt;
}
```

Cochez la ou les instructions correctes pour remplacer la ligne `/* initialisation du champ data */` :

☐
```
pelt->data=data;
```

☐
```
pelt->data=&data;
```

☐
```
pelt->data=*data;
```

■
```
pelt->data=malloc(sizeof(int));
*(pelt->data) = data;
```

☐
```
pelt->data=malloc(sizeof(int));
pelt->data = data;
```

Soit la fonction suivante :

```
void map(PElement liste, void (*ma_fonction)(void *data)) {
   while(liste) {
      ma_fonction(liste->data);
      liste=liste->suivant;
   }
}
```

Question 13 (1 point)

On souhaite afficher le contenu d'une liste d'entiers avec la fonction `map` ci-dessus. Écrivez la fonction `afficher_entier` à transmettre en argument lors de l'appel à `map`. Vous afficherez juste l'entier suivi d'un espace.

Solution:

```
void afficher_entier(void *data) {
   int *di=(int *)data;
   printf("%d ",*di);
}
```

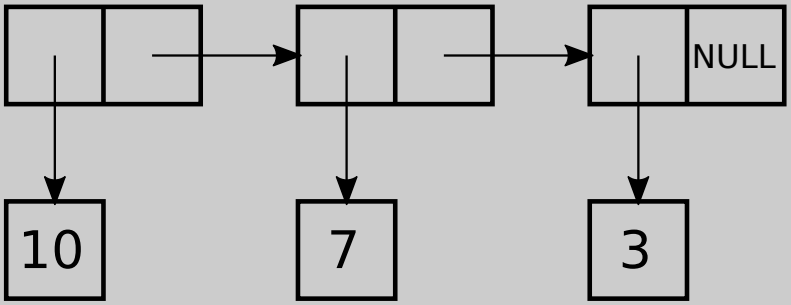
Soit le code suivant :

```
PElement pelt=creer_elt_entier(10);
pelt->suivant=creer_elt_entier(7);
pelt->suivant->suivant=creer_elt_entier(3);
```

Question 14 (1 point)

Quel est l'espace mémoire occupé par cette liste (architecture 32 bits) ? Vous dessinerez cette structure en indiquant pas des flèches sur quoi pointent les pointeurs. Pour rappel, un `int` et un pointeur prennent tous les deux 4 octets sur ce type d'architecture de processeur.

Solution: Il y a 6 pointeurs et 3 entiers, donc 9x4=36 octets.



Question 15 (1 point)

Nous voulons maintenant afficher la liste précédente en utilisant la fonction `map`. Ecrivez la ou les instructions le permettant et indiquez ce qui sera affiché à l'écran.

Solution:

```
map(pelt,afficher_entier);
```

Affichage à l'écran :

```
10 7 3
```

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`.