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

Programmation et structures de données en C– LU2IN018

Partiel du 3 décembre 2020

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 27 points (13 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)

Soit l'instruction suivante :

```
int *t=(int *)malloc(sizeof(int)*10);
```

Cochez la ou les réponses correctes :

- ☐ `free(t)` permet de libérer toute la mémoire allouée
- ☐ `free(*t)` permet de libérer toute la mémoire allouée
- ☐ `free(&t)` permet de libérer toute la mémoire allouée
- ☐ il est possible de ne libérer qu'une partie de la mémoire allouée

Question 2 (2 points)

Soit une liste chaînée de messages déclarée avec la structure suivante :

```
typedef struct _elt {
  char *msg;
  struct _elt *suiv;
} Elt;
```

Écrivez la fonction de création d'un élément. Prototype :

```
Elt *creer_elt(const char *msg);
```

La chaîne pointée par l'argument `msg` est susceptible d'être libérée juste après cet appel. Une fois créé, cet élément doit pouvoir être inséré en queue d'une liste chaînée sans modification des champs de la structure.

Réponse :

Question 3 (2 points)

Une liste de 10 messages a été créée. La liste chaînée et les données qu'elle contient occupent en mémoire :

- ☐ 10 fois l'espace nécessaire pour stocker un pointeur et l'espace nécessaire pour stocker les chaînes de caractères
- ☐ 10 fois l'espace nécessaire pour stocker deux pointeurs et l'espace nécessaire pour stocker les chaînes de caractères
- ☐ 10 fois l'espace nécessaire pour stocker deux pointeurs
- ☐ uniquement l'espace occupé par les chaînes de caractères
- ☐ 2 fois l'espace occupé par chaque chaîne de caractères

Soit la fonction de prototype :

```
void f(int **l, int *m);
```

Question 4 (2 points)

Cochez la ou les réponses correctes :

- ☐ Le paramètre `l` permet de transmettre un tableau de pointeurs sur entiers
- ☐ Le paramètre `l` permet de transmettre un entier (non pointeur) par copie
- ☐ Le paramètre `l` permet de transmettre un pointeur que l'on souhaite modifier
- ☐ l'affectation `l=NULL;` permet de faire une modification non locale à `f`
- ☐ l'affectation `*l=NULL;` permet de faire une modification non locale à `f`

Question 5 (2 points)

Cochez la ou les réponses correctes :

- ☐ Le paramètre `m` permet de transmettre un tableau d'entiers
- ☐ Le paramètre `m` permet de transmettre un entier (non pointeur) par copie
- ☐ Le paramètre `m` permet de transmettre un pointeur que l'on souhaite modifier
- ☐ l'affectation `m=NULL;` permet de faire une modification non locale à `f`
- ☐ l'affectation `*m=0;` permet de faire une modification non locale à `f`

Soit le code suivant stocké dans le fichier `bug.c` :

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  typedef struct _Objet {
6    int id;
7    char nom[100];
8    char *description;
9  } Objet;
10
11 Objet *creer_objet(int id, char *nom, char *description) {
12   Objet *no=malloc(sizeof(Objet));
13   no->id=id;
14   strcpy(no->nom,nom);
15   strcpy(no->description, description);
16   return no;
17 }
18
19 void afficher_objet(Objet *obj) {
20   printf("Id: %d, nom: %s\nDescription: %s\n", obj->id, obj->nom, obj
       ->description);
21 }
22
23 int main(void) {
24
25   char *des=strdup("Beau, pas cher, utile, c'est l'objet qu'il vous
       faut");
26   Objet *o=creer_objet(155, "Mon_super_objet", des);
27   afficher_objet(o);
28   free(des);
29   free(o);
30   return 0;
31
32 }
```

Il a été compilé de la façon suivante et exécuté :

```
bash$ gcc -g -Wall -o bug1 bug.c
bash$ ./bug
Segmentation fault (core dumped)
```

Question 6 (2 points)

Cochez la ou les réponses correctes :

- ☐ C'est un problème de compilation
- ☐ C'est un problème d'exécution
- ☐ `ddd` peut aider à résoudre ce problème
- ☐ `gcc` peut aider à résoudre ce problème

Question 7 (2 points)

L'exécution de ce code avec valgrind donne le résultat suivant :

```
$ valgrind ./bug1
==2646== Memcheck, a memory error detector
==2646== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2646== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2646== Command: ./bug1
==2646==
==2646== Use of uninitialised value of size 8
==2646==    at 0x4C32E00: strcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.
==2646==    by 0x108771: creer_objet (bug.c:15)
==2646==    by 0x1087DD: main (bug.c:26)
==2646==
==2646== Invalid write of size 1
==2646==    at 0x4C32E00: strcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.
==2646==    by 0x108771: creer_objet (bug.c:15)
==2646==    by 0x1087DD: main (bug.c:26)
==2646==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==2646==
==2646==
==2646== Process terminating with default action of signal 11 (SIGSEGV)
==2646==  Access not within mapped region at address 0x0
==2646==    at 0x4C32E00: strcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.
==2646==    by 0x108771: creer_objet (bug.c:15)
==2646==    by 0x1087DD: main (bug.c:26)
==2646==  If you believe this happened as a result of a stack
==2646==  overflow in your program's main thread (unlikely but
==2646==  possible), you can try to increase the size of the
==2646==  main thread stack using the --main-stacksize= flag.
==2646==  The main thread stack size used in this run was 8388608.
==2646==
==2646== HEAP SUMMARY:
==2646==     in use at exit: 165 bytes in 2 blocks
==2646==   total heap usage: 2 allocs, 0 frees, 165 bytes allocated
==2646==
==2646== LEAK SUMMARY:
==2646==    definitely lost: 0 bytes in 0 blocks
==2646==    indirectly lost: 0 bytes in 0 blocks
==2646==      possibly lost: 0 bytes in 0 blocks
==2646==    still reachable: 165 bytes in 2 blocks
==2646==         suppressed: 0 bytes in 0 blocks
==2646== Rerun with --leak-check=full to see details of leaked memory
==2646==
==2646== For counts of detected and suppressed errors, rerun with: -v
==2646== Use --track-origins=yes to see where uninitialised values come from
==2646== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

Que pouvez-vous déduire de ces indications ? Où se situe le problème ? Quel est-il ?

Réponse :

Question 8 (2 points)

Proposez une solution au problème

Réponse :

Un objet est écrit de la façon suivante dans un fichier texte :

```
155
Mon_super_objet
Beau, pas cher, utile, c'est l'objet qu'il vous faut
```

Soit la fonction de lecture suivante :

```
1 Object *lire_objet(FILE *f) {
2   char nom[100];
3   char description[200];
4   int id;
5   /* lecture de id */
6   /* lecture du nom */
7   /* lecture de la description */
8   return creer_objet(id,nom, description);
9 }
```

Question 9 (2 points)

Indiquez la ou les instructions correctes pour la lecture de l'id (ligne 5). Vous ne vous préoccuperez pas des retours à la ligne.

☐ `fread(&id, sizeof(int), 1, f);`

☐ `fread(id, sizeof(int), 1, f);`

☐ `fscanf(f, "␣%d␣", id);`

☐ `fscanf(f, "␣%d␣", &id);`

☐ `id=fgetc(f);`

☐ `&id=fgetc(f);`

☐ `fgets(id, 10, f);`

☐ `fgets(&id, 10, f);`

Question 10 (2 points)

Indiquez la ou les instructions correctes pour la lecture du nom (ligne 6). Vous ne vous préoccuperez pas des retours à la ligne et nous ferons l'hypothèse que le nom d'un objet ne contient pas d'espaces.

☐ `fread(&nom, sizeof(char), 100, f);`

☐ `fread(nom, sizeof(char), 100, f);`

☐ `fscanf(f, "␣%s␣", nom);`

☐ `fscanf(f, "␣%s␣", &nom);`

☐ `nom=fgetc(f);`

☐ `&nom=fgetc(f);`

☐ `fgets(nom, 100, f);`

☐ `fgets(&nom, 100, f);`

Question 11 (2 points)

Soit la fonction d'écriture suivante :

```
1 void ecrire_objet(FILE *f, Objet *obj) {
2   /* ecriture de l'objet */
3 }
```

Indiquez la ou les instructions correctes pour écrire l'objet selon le format indiqué ci-dessus.

☐ `fputc(&obj,f);`

☐ `fputc(obj,f);`

☐ `fwrite(&obj, sizeof(Objet), 1, f);`

☐ `fwrite(obj, sizeof(Objet), 1, f);`

☐ `fprintf(f,"%d\n%s\n%s\n",&obj->id, &obj->nom, &obj->description);`

☐ `fprintf(f,"%d\n%s\n%s\n",obj->id, obj->nom, obj->description);`

Question 12 (3 points)

Ecrivez une fonction main qui ouvre le fichier "objet_source.txt", lit l'objet qu'il contient et l'écrit dans un fichier "objet_destination.txt". Vous ferez l'hypothèse que l'ouverture du fichier et la lecture se passent bien sans le vérifier. Vous prendrez soin de libérer toute la mémoire allouée.

Réponse :

Question 13 (2 points)

Les fonctions précédentes sont réparties entre deux fichiers : `objet.c` et `main_objet.c`. Les prototypes des fonctions et la déclaration de structure est dans `objet.h`. Indiquez la ou les façons de créer un exécutable à partir de ces fichiers :

☐ `ddd -Wall -o main_objet main_objet.c objet.c`

☐
```
ddd -c -Wall -o main_objet.o main_objet.c
ddd -c -Wall -o objet.o objet.c
ddd -Wall -o main_objet main_objet.o objet.o
```

☐ `ddd -Wall -o main_objet main_objet.c objet.h`

☐ `ddd -Wall -o main_objet main_objet.c`

☐
```
gcc -c -Wall -o main_objet.o main_objet.c
gcc -c -Wall -o objet.o objet.c
gcc -Wall -o main_objet main_objet.o objet.o
```

☐ `gcc -Wall -o main_objet main_objet.c objet.c`

☐ `gcc -Wall -o main_objet main_objet.c objet.h`

☐ `gcc -Wall -o main_objet main_objet.c`

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`.