NOM :
Prénom :

Examen de SELC/INF104

1h30 – Sans document

Nous avons laissé de la place sur l'énoncé pour vos réponses. Vous pouvez donc rendre l'énoncé complété. Attention, ***mettez votre nom sur toutes les feuilles rendues.***

Questions de cours (12 points)

Bases du langage C (3 points)

Question 1 (1 point) : On considère le code suivant :

```
int a = 5;
int * ptr = __&a__ ;
int b = ___*ptr___;
```

Complétez ce code pour initialiser ptr avec l'adresse de la variable a ; et initialiser b avec la valeur pointée par ptr.

Question 2 (0,5 point) : Décrivez le fonctionnement de l'opérateur sizeof en C

Sizeof est un opérateur qui s'utilise comme une fonction: il prend en paramètre un type de donnée ou une variable et renvoie sa taille en octets. Cependant, ce n'est pas une fonction mais un opérateur: la valeur de retour est déterminée au moment de la compilation et pas lors de l'exécution du programme.
Exemple d'utilisation: sizeof(char) renvoie 1.

On considère le code suivant :

```
int main(int argc, char*argv[])
{
  char * name1 = "Adrien";
  char * name2 = "Adrien";

  if(name1==name2)
  {
    printf("Is this ok?\n");
  }
  return 0;
}
```

Question 3 (0,5 point) : Quel sera l'affichage résultant de l'exécution de ce programme ? Pourquoi ?

Réponse attendue: name1 et name2 correspondent à des adresses différentes donc le programme n'affiche rien.

Ce que gcc fait vraiment (réponse acceptée donc mais pas vraiment attendue): name1 et name2 référencent la même adresse dans la section des données en lecture seul du programme (grâce à une optimisation du compilateur); donc l'exécution programme affiche "Is this ok?"

Réponse non acceptée: le programme compare le contenu des chaines de caractère et affiche "Is this ok?" car le contenu des deux chaines est le même.

NOM :
Prénom :

Question 4 (0,5 point) : Considérons un pointeur `ptr` et un entier `k`. Ces deux notations sont elles équivalentes : `*(ptr+k)` et `ptr[k]` ? Justifiez votre réponse

Ces notations sont bien équivalentes.
ptr[k] renvoie la valeur stockée dans un tableau commençant à l'adresse ptr et stockant des éléments de type *ptr. ptr[k] est les k+1ème élément de ce tableau.
ptr+k correspond à l'adresse ptr+k*sizeof(*ptr). Les adresses étant contiguës en mémoire, il s'agit de la case mémoire d'adresse ptr+un décalage de k case mémoire de taille (*ptr).
*(ptr+k) dé-référence cette case mémoire et renvoie donc la même valeur que ptr[k].

Question 5 (0,5 point) : Comment sont initialisées les valeurs des paramètres argc et argv de la fonction main ? Donnez un exemple.

argc est un entier initialisé avec le nombre d'arguments passés au programme sur la ligne de commande
argv est un tableau de chaines de caractères, et chaque case de ce tableau est initialisé avec une des chaines de caractère passé en argument au programme sur la ligne de commande. L'ordre d'arrivé des arguments sur la ligne de commande est conservé dans ce tableau.

Exemple: ./appli arg1 arg2 —> argc vaut 3, argv[0] vaut "./appli", argv[1] vaut "arg1" et argv[2] vaut "arg2".

Bases de la chaine de production (4 points)

On dispose d'une machine 64 bits. Soit le fichier source « appli.c » ci-dessous. La fonction "racine" renvoie la racine carrée et fonctionne correctement, `2.25e-308` est la valeur la plus petite d'un nombre flottant positif encodé sur 64 bits.

appli.c

```
#include <stdio.h>
#define MINDIFF 2.25e-308

float racine(float carre)
{   float resultat=carre/3, ancien, difference=1;
    if (carre <= 0) return 0;
    do {ancien = resultat;
        resultat = (resultat + carre / resultat) / 2;
        difference = resultat - ancien;
    } while (difference > MINDIFF || difference < -MINDIFF);
    return resultat;
}
float carre(float x)
{   return x*x;
}
int main(int argc, char* argv[])
{   float ax = 3.0, ay = 5.0;
    float bx = 7.0, by = 8.0;
    float distance = racine(carre(bx-ax)+carre(by-ay));
    printf("distance = %f\n",distance);
    return 0;
}
```

Question 6 (1 point) : Le programme est-il compilable sur cette machine ? Quelle sera la trace d'exécution de ce programme ?

Le programme est compilable; trace d'exécution: distance = 5.0

Question 7 (0,5 point) : Quelle est la commande à taper dans un terminal pour compiler, en affichant tous les Warning éventuels, et produire un exécutable nommé « appli » ?

gcc -Wall appli.c -o appli

NOM :
Prénom :

Question 8 (0,5 point) : Quelle est la commande à taper pour exécuter « appli » ?

./appli

Question 9 (0,5 point) : A quoi correspond la ligne `#define MINDIFF 2.25e-308` ? Quel est son effet ?

Il s'agit d'une directive au préprocesseur: avant la compilation du code source C, le préprocesseur remplacera toutes les occurrences de MINDIFF dans le code par 2.25e-308.

On déplace les fonctions carre et racine dans un nouveau fichier source :

fmath.c

```
#define MINDIFF 2.25e-308

float racine(float carre)
{   float resultat=carre/3, ancien, difference=1;
    if (carre <= 0) return 0;
    do {ancien = resultat;
        resultat = (resultat + carre / resultat) / 2;
        difference = resultat - ancien;
    } while (difference > MINDIFF || difference < -MINDIFF);
    return resultat;
}
float carre(float x)
{   return x*x;
}
```

Question 10 (1 point) : Ecrire le fichier fmath.h avec les signatures des fonctions du fichier fmath.c

fmath.h

```
#ifndef _FMATH_H_
#define _FMATH_H_

float racine(float carre);
float carre(float x);

#endif
```

Question 11 (0,5 point) : Compléter le fichier appli.c pour prendre en compte ce déplacement.

appli.c

```
#include <stdio.h>
#include "fmath.h"


int main(int argc, char* argv[])
{   float ax = 3.0, ay = 5.0;
    float bx = 7.0, by = 8.0;
    float distance = racine(carre(bx-ax)+carre(by-ay));
    printf("distance = %f\n",distance);
    return 0;
}
```

NOM :
Prénom :

Compilation séparée et Makefile (5 points)

Question 12 (0,5 point) :

Quelle est la commande à taper dans un terminal pour compiler, en affichant les Warning éventuels, et produire un fichier objet fmath.o à partir d'un fichier fmath.c ?

```
gcc -Wall -c fmath.c
```

Quelle est la commande à taper dans un terminal pour compiler, en affichant les Warning éventuels, et produire un fichier objet appli.o à partir d'un fichier appli.c ?

```
gcc -Wall -c appli.c
```

Question 13 (0,5 point) : Quelle est la commande à taper dans un terminal pour produire un exécutable nommé « appli » à partir des fichiers objets fmath.o et appli.o ?

```
gcc fmath.o appli.o -o appli
```

Question 14 (2,5 points) : Ecrire un makefile simple qui automatise les commandes des questions 12 et 13. Le makefile n'utilisera pas de variables et se réduira à sa plus simple expression. Il contiendra trois cibles:
- la cible pour produire fmath.o
- la cible pour produire appli.o
- la cible pour produire appli

```
fmath.o: fmath.c fmath.h
	gcc -Wall -c fmath.c

appli.o: appli.c fmath.h
	gcc -Wall -c appli.c

appli: fmath.o appli.o
	gcc -Wall fmath.o appli.o
```

NOM :
Prénom :

Question 15 (1,5 points): On suppose que la commande "ls -l" renvoie les informations suivantes, qui incluent les dates de dernières modifications des différents fichiers :

appli.c	27/01/2016 – 11:25
appli.o	27/01/2016 – 10:37
appli	27/01/2016 – 10:37
fmath.c	27/01/2016 – 10:30
fmath.o	27/01/2016 – 10:35
Makefile	27/01/2016 – 09:59

Décrire ce qu'il se passe, dans ces conditions, si l'on tape dans un terminal la commande : make appli

appli.c a été modifié plus récemment que appli.o
La cible appli.o est donc reconstruite : compilation du fichier appli.c.
la cible appli est ensuite reconstruite : édition de lien entre le nouveau appli.o et l'ancien fmath.o

Problème (8 points)

On souhaite implanter HEAP, un système efficace d'allocation dynamique d'objets de type object_t. Cette efficacité est obtenue en allouant à l'avance de la mémoire à des objets, ceci afin d'éviter de le faire à chaque demande d'allocation, ce qui peut avoir un coût d'exécution. La première idée consiste à rendre disponibles, dès l'initialisation, **MIN** objets en leur allouant une place en mémoire. La deuxième idée consiste à sauvegarder, dans une liste chaînée, jusqu'à **HIGH** objets, que l'on aurait dû détruire à la demande de l'utilisateur, pour les réutiliser lors de demandes d'allocation ultérieures. Cependant, si le nombre d'objets ainsi sauvegardés devient supérieur à **HIGH**, on détruit effectivement leur place en mémoire. La troisième idée garantit de ne jamais allouer en mémoire plus de **MAX** objets. On suppose **MIN <= HIGH <= MAX**. Le type object_t défini par l'utilisateur ne permet pas de relier les objets sauvegardés, plus exactement les places en mémoire des objets sauvegardés, sous forme de liste chaînée. On va définir un type node_t pour relier ces places en mémoire les unes avec les autres. On va utiliser un « mécanisme » du langage C pour considérer une même place en mémoire sous forme de structures complètement différentes. Dans la suite, ce mécanisme va permettre de considérer une même place en mémoire comme une structure de type object_t mais aussi de type node_t.

Les spécifications du système d'allocation sont les suivantes (lire attentivement).

A. La liste chaînée **unusedNodes** représente la liste d'objets auxquels a été allouée une place en mémoire et que l'on sauvegarde pour satisfaire des demandes ultérieures.
B. Le compteur **inMemoryNodes** représente le nombre d'objets auxquels a été allouée une place en mémoire. Il comptabilise les objets fournis par HEAP et en cours d'utilisation dans le reste du programme et les objets que l'on aurait du détruire mais que l'on a sauvegardés dans la liste chainée **unusedNodes**. Il faudra veiller à maintenir correcte la valeur de ce compteur.
C. HEAP dispose d'une fonction d'initialisation ***newHeap()*** qui alloue à l'avance des places en mémoire à **MIN** objets qu'elle place dans **unusedNodes** (voir A).
D. HEAP n'alloue jamais plus en mémoire de **MAX** objets, donc **inMemoryNodes** n'est jamais supérieur à **MAX**.

NOM :
Prénom :

E. La fonction ***allocate()*** de HEAP alloue un nouvel objet si **unusedNodes** (voir A) est vide et si **inMemoryNodes** (voir B) n'est pas supérieur à **MAX**.
F. La fonction ***allocate()*** retourne un objet qu'elle aura extrait de **unusedNodes** (voir A) si **unusedNodes** n'est pas vide.
G. La fonction ***destroy()*** insère l'objet qu'elle devait détruire dans ***unusedNodes*** (voir A) pour le réutiliser lors d'une prochaine allocation si **inMemoryNodes** (voir B) est inférieur à **HIGH**.
H. La fonction ***destroy()*** libère, grâce aux fonctions du gestionnaire de mémoire dynamique du langage C, la place en mémoire de l'objet qu'elle doit détruire si **inMemoryNodes** (voir B) est supérieur ou égal à **HIGH**.

Vous pouvez implanter la liste chainée ***unusedNodes*** comme en TP sous forme de FIFO (First In, First Out). Cependant, il est bien plus facile de traiter ce problème en utilisant une LIFO (Last In, First Out). Dans une LIFO, lorsque vous insérez un nouvel élément, il se trouve placé au début de la liste. Lorsque vous retirez un élément, vous le prenez au début de la liste.

Vous répondrez aux questions (sauf pour 18 et 23) dans le listing de code de la feuille suivante.

Question 16 (1 point) : Compléter la définition du type *node_t* pour que celui-ci implémente une liste chainée (commentaire 1.16). *node_t* sert à implanter *unusedNodes*. *node_t* ne contient qu'un seul attribut *next* qui référence le nœud suivant dans la liste chainée.

Question 17 (1 point) : Compléter *newObject()* (commentaire 1.17.1) et *newHeap()* (commentaire 1.17.2) pour que ces fonctions effectuent l'allocation de *object_t* et *heap_t* en utilisant les fonctions du gestionnaire de mémoire dynamique du langage C. La compilation se fera **sans** *Warning.*

Question 18 (1 point) : Expliquer en quoi consistent dans le langage C les opérations « ***(object_t *)*** » (commentaire 1.18.1) ou « ***(node_t *)*** » (commentaire 1.18.2). Indiquer ce que ces « mécanismes » permettent de faire dans le contexte de l'implantation de HEAP.

Question 19 (1 point) : Compléter *newHeap()* pour que MIN nœuds soient présents dans la liste *unusedNodes*, comme indiqué en spécifications C (commentaire 1.19).

Question 20 (1 point) : Compléter *allocate()* pour satisfaire la spécification F (commentaire 1.20).

Question 21 (1 point) : Compléter *destroy()* pour satisfaire les spécifications B et G (commentaire 1.21).

Question 22 (1 point) : Compléter *destroy()* pour satisfaire les spécifications B et H (commentaire 1.22).

NOM :
Prénom :

Question 23 (1 point) : On modifie la signature de *allocate()* pour qu'elle retourne le nombre de nœuds alloués en mémoire comme le fait actuellement *destroy().* On propose la signature suivante :

int allocate(heap_t * heap, object_t * object) ;

Expliquer pourquoi cette signature pose problème et donner la signature correcte.

NOM :
Prénom :

NOM :
Prénom :

```
#define MIN  20
#define HIGH 40
#define MAX  60

typedef struct {long x, y ;} object_t;

typedef struct node                          {                    // comment 1.16
    struct node                          * next;
} node_t;

typedef struct {
  node_t   * unusedNodes;
  unsigned inMemoryNodes;
} heap_t;

object_t * newObject(){
  return      (Object *)malloc(sizeof(Object))                   // comment 1.17.1
}

heap_t * newHeap(){
  node_t * node;
  heap_t * heap = (Heap *)malloc(sizeof(Heap))                   // comment 1.17.2
  heap->unusedNodes = NULL;
  for(heap->inMemoryNodes=0; heap->inMemoryNodes<MIN; heap->inMemoryNodes++)
  {
    node = (node_t *)newObject();                                // comment 1.18.1
    node->next = heap->unusedNodes ;                             // comment 1.19
    heap->unusedNodes = node ;                                   //
                                                                 //
  }
  return heap;
}

object_t * allocate(heap_t * heap){
  node_t * node;
  if (heap->unusedNodes == NULL) {
    if (heap->inMemoryNodes == MAX) return NULL;
    heap->inMemoryNodes++;
    return newObject();
  }
   node = heap->unsedNodes ;                                     // comment 1.20
   heap->unusedNodes = heap->unsedNodes->next ;                  //
                                                                 //
  return (object_t *)node;                                       // comment 1.18.2
}

int destroy(heap_t * heap, object_t * object){
  node_t * node = (node_t *) object ;
  if (heap->inMemoryNodes < HIGH){
     node->next = heap->unusedNodes ;                            // comment 1.21
     heap->unsedNodes = node ;                                   //
                                                                 //
  } else {
     free(node) ;                                                // comment 1.22
     heap->nCreatedNodes-- ;                                     //
                                                                 //
  }
  return heap->inMemoryNodes;
}
```