Examen de SELC/INF104

1h30 – Sans document

Questions de cours (14 points)

Bases du langage C (6 points)

Question 1 (1 point) : Décrivez le fonctionnement de l'opérateur sizeof en C

Question 2 (1 point) : Considérons un pointeur `ptr` et un entier `k`. Ces deux notations sont elles équivalentes : `*(ptr+k)` et `ptr[k]` ? Justifiez votre réponse.

Question 3 (4 points) : On considère le code suivant, dans lequel vous allez faire la chasse aux erreurs de programmation. Il y a un point par erreur identifiée, expliquée, et résolue. Les questions qui suivent vous aideront à identifier les erreurs :

```c
#include <stdio.h>

int * len;

int main(int argc, char*argv[]) {
  int ret = stringlen(argv[1]);
  char copy[8];
  int i;
  for(i=0; i=ret-1; i++) {
    copy[i]=argv[1][i];
  }
  printf("Taille de la chaine à copier : %s\n", ret);
  printf("copy : %s\n", copy);
}

int stringlen(char * str) {
  int i = 0;
  while(str[i]!='\0') {
    i++;
  }
  return i;
}
```

Explications : ce programme doit récupérer la longueur de la chaine de caractères `argv[1]`, puis copier cette chaine dans la variable `copy`, puis afficher la longueur et la copie de la chaine,.

Question 3.1 (1 point): Le code étant contenu dans un fichier `main.c`, on compile le programme en exécutant la commande `gcc main.c -o appli`. On voit alors le message suivant sur le terminal:

```
main.c:5:13: warning: implicit declaration of function 'stringlen'
  int ret = stringlen(argv[1]);
```

Expliquez pourquoi et proposez une résolution

Question 3.2 (1 point): On suppose qu'on a résolu le problème précédent. On exécute le programme avec la commande suivante : `./appli fic1`. On observe alors que rien ne s'affiche et que le programme ne se termine jamais. Expliquez pourquoi et proposez une résolution.

Question 3.3 (1 point) : On suppose qu'on a bien résolu le problème précédent. La compilation se déroule sans alerte. On exécute le programme avec la ligne suivante : `./appli fic1` Le programme affiche seulement : « `Segmentation fault` », ce qui correspond à une erreur de segmentation: ceci signifie que le programme essaie d'accéder à une zone mémoire qui n'a pas été réservée pour son exécution. La faute de programmation se trouve à la ligne 12, dans l'appel à la fonction `printf("Taille de la…)` . Expliquez pourquoi l'exécution de cette ligne provoque une erreur de segmentation et proposez une résolution.

Question 3.4 (1 point) : On suppose les problèmes précédents résolus. Cependant, un dernier problème demeure : la taille du tableau `copy` est de 8 éléments de type `char`. C'est très limitant, et on ne sait pas à l'avance quelle sera la longueur des arguments passés sur la ligne de commande. Identifiez une fonction système (présente dans la bibliothèque standard du langage C) qui permette d'avoir un tableau dont la taille dépend du résultat de l'appel à la fonction `stringlen`. Proposez une modification du code ci-dessus qui utilise la fonction que vous aurez identifiée. N'oubliez pas d'inclure le fichier en-tête (.h) contenant la signature de cette fonction.

Chaine de production (7 points)

On donne, ci dessous le contenu de 4 fichiers : `main.c`, `math.h`, `dist.h`, et `dist.c`

Fichier `main.c`

```
#include <math.h>
#include "dist.h"

#define X_OFFSET 0.0
#define Y_OFFSET 0.0

int main(int argc, char* argv[]) {
  point_t p1 = {3.0-X_OFFSET,5.0-Y_OFFSET};
  point_t p2 = {7.0-X_OFFSET,8.0-Y_OFFSET};
  double dist = distance(p1, p2);
#ifdef DEBUG
  printf("distance = %f\n",dist);
#endif
  return 0;
}
```

Fichier `math.h`

```
#ifndef __MATH_H__
#define __MATH_H__

double sqrt(double x);
double pow(double x, double y);

#endif
```

Fichier `dist.h`

```
#ifndef __DIST_H__
#define __DIST_H__

#include <math.h>

typedef struct point {
  double x;
  double y;
} point_t;

double distance(point_t p1, point_t p2);

#endif
```

Fichier `dist.c`

```
#include "dist.h"
#include <math.h>

double distance(point_t p1, point_t p2) {
  return sqrt(pow(p2.x-p1.x, 2.0)+pow(p2.y-p1.y,2.0));
}
```

Question 4 (3 points): Nous nous intéressons dans cette question à l'effet du préprocesseur (outil `cpp` utilisé par `gcc` dans la première étape de compilation d'un programme C). L'option `-E` de `gcc` permet d'afficher sur le terminal le contenu du fichier produit par le préprocesseur à partir d'un fichier source.

Ecrivez ce qui s'affiche sur le terminal par exécution de la commande : `gcc -E main.c`. Il est recommandé d'expliquer votre réponse.

Question 5 (4 points) : Ecrire un fichier `Makefile`, qui automatise la production d'un fichier exécutable `appli` à partir des fichiers `main.c`, `dist.c`, `dist.h` et `math.h`. Le fichier `Makefile` restera simple : vous n'avez PAS BESOIN D'UTILISER des variables ni des mécanismes sophistiqués (i.e. règles de préfix, mise à jour automatique des dépendances, etc.). Votre `Makefile` contiendra trois cibles:

- la cible pour produire `dist.o` à partir de `dist.c`
- la cible pour produire `main.o` à partir de `main.c`
- la cible pour produire `appli` à partir de `main.o`, `dist.o`, et de la librairie `math` (contenant les implémentations binaires des fonctions `sqrt` et `pow`) : `libm.a`. On supposera que `gcc` connaît l'emplacement de cette librairie sur le disque.

INDICATION : si vous ne savez pas écrire le contenu du fichier `Makefile`, indiquez quand même les commandes permettant de faire chacune des étapes de la compilation séparée, telles que décrites dans cette question. Cela vous rapportera une partie des points…

Problème (7 points)

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct vertex {
  char label ;
  struct vertex ** neighbours ;
  int edges_nb ;
  int unique_identifier;
} vertex_t ;

void add_neighbours(vertex_t * v, vertex_t ** neighbours, int
nb_of_neighbours)
{
  /* Question 6.1 */
}

vertex_t * create_vertex(char label)
{
  /* Question 6.2 */
}



void display_vertex_label(vertex_t v)
{
  printf("vertex label : %c\n", v.label);
}



char marked[10]; /*Question 6.3*/

void dfs(vertex_t v)
{
  marked[v.unique_identifier] = 1;
  int i;
  for(i=0; i<v.edges_nb;i++) {
    vertex_t * neighbour = v.neighbours[i];
    if(neighbour != NULL)
    {
      if(marked[neighbour ->unique_identifier]!=1){
        dfs(*neighbour);
      }
    }
  }
  display_vertex_label(v);
}
```

```c
int main(int argc, char * argv[])
{
  vertex_t * v_a = create_vertex('a');
  vertex_t * v_b = create_vertex('b');
  vertex_t * v_c = create_vertex('c');
  vertex_t * v_d = create_vertex('d');
  vertex_t * v_e = create_vertex('e');

  vertex_t * graph[5] = {v_a,v_b,v_c,v_d,v_e} ;

  vertex_t * a_neighbours[1] = {v_b};
  vertex_t * c_neighbours[3] = {v_b,v_d,v_e};
  vertex_t * d_neighbours[2] = {v_b,v_e};
  vertex_t * e_neighbours[1] = {v_c};

  add_neighbours(v_a, a_neighbours, 1);
  add_neighbours(v_c, c_neighbours, 3);
  add_neighbours(v_d, d_neighbours, 2);
  add_neighbours(v_e, e_neighbours, 1);

  dfs(*v_c);

}
```

Question 6 (3 points) : Le code ci-dessus propose une implémentation d'un algorithme classique de parcours de graphe : le parcours en profondeur (*depth first search*). Dans le code qui vous est fourni, certaines fonctions sont à compléter ou à améliorer. Les questions ci-dessous vous guideront dans ce travail.

Explications concernant le code fourni.

Dans ce code, la fonction `dfs` implémente un parcours en profondeur du graphe. Un graphe est ici représenté sous la forme d'une liste d'adjacence : pour chaque nœud, ses voisins sont stockés dans une liste de nœuds. Le code de la fonction main initialise un graphe de 5 nœuds. Le type de donnée associé à un nœud est fourni au début du code. Il s'agit du type `vertex_t`. Ce type de donnée est une structure contenant :

- un champ label associant une information utile au nœud. Nous considérons ici de simples caractères permettant d'identifier facilement un nœud. Dans le code de la fonction `main`, 5 nœuds sont créés en faisant appel à la fonction `create_vertex` (à compléter en réponse à la question 6.2). Leurs labels sont a, b, c, d, et e.
- un champ `neighbours`, qui correspond à la liste des voisins du nœud dans la liste d'adjacences décrivant le graphe. Par exemple, le nœud c aura trois voisins : b, d, et e. Chaque élément du tableau `neighbours` correspond à une adresse: l'adresse d'un noeud voisin. Le champ

`neighbours` du nœud c sera donc initialisé avec un tableau contenant les adresses des nœuds b, d, et e. Cette initialisation sera effectuée dans la fonction `add_neighbours` (à compléter en réponse à la question 6.1).

- un champ `edges_nb`, de type `int`, qui contient le nombre de voisins du nœud.
- un champ `unique_identifier`, de type `int`, qui servira d'identifiant du nœud pour vérifier rapidement si celui-ci a déjà été traité lors du parcours en profondeur. L'ensemble des nœuds traités sera stocké dans le tableau `marked` (variable globale). L'identifiant d'un nœud correspondra à un élément du tableau. Pour un élément du tableau, la valeur 1 signifiera que le nœud correspondant a déjà été traité par la fonction `dfs`.

Notez que l'initialisation des nœuds se fait en deux temps car il faut d'abord créer un nœud pour pouvoir ensuite le référencer comme voisin d'un autre nœud. Nous créons d'abord tous les nœuds sans initialiser la liste de leurs voisins avec la fonction `create_vertex`. Ensuite, nous initialisons la liste des voisins de chaque nœud créé (sauf si un nœud n'a pas de voisin comme le nœud b par exemple) avec la fonction `add_neighbours`.

Les questions suivantes sont classées par ordre de difficulté croissante :

Question 6.1 (1 point) : écrivez le code de la fonction `add_neighbours`. Cette fonction initialise les champs `neighbours` et `edges_nb` d'un nœud dont l'adresse est passée en paramètre de la fonction. L'adresse du nœud en question est identifié par le paramètre `v`. En lisant le code de la fonction `main` et les explications ci-dessus, complétez le code de la fonction `add_neighbours`.

Question 6.2 (2 points) : écrivez le code de la fonction `create_vertex`, qui crée un nouveau nœud et initialise ses champs à partir des paramètres de la fonction. Si aucun paramètre ne correspond à un des champs de la structure, initialisez le avec une valeur qui vous paraît correspondre à l'usage du champ en question dans la fonction `dfs`.

Indication : vous pouvez ajouter une variable globale ou une variable locale statique pour répondre à cette question.

Question 7 (4 points) : un collègue vous fait la suggestion suivante : « la solution que nous avons implémenté occupe inutilement de l'espace mémoire : un tableau de `char` sert à stocker un ensemble d'informations booléennes (vrai ou faux) alors qu'une case mémoire représente déjà un ensemble d'informations booléennes (1 ou 0). » Nous pensons qu'il a raison, et nous vous proposons d'utiliser une représentation binaire pour la variable globale `marked`.

Pour vous aider, nous rappelons ici le fonctionnement des opérateurs binaires (|, &, <<, et >>) :

« | » est un OU logique bit à bit. Exemple : 01100101 | 00011011 = 01111111	« & » est un ET logique bit à bit. Exemple : 01100101 & 00011011 = 00000001
« << N » est un décalage de N bits vers la gauche. Exemple : 01100101 << 4 = 01010000	>> N est un décalage de N bits vers la droite. Exemple : 01100101 >> 4 = 00000110

Dans le code que vous avez écrit, et dans le code qui vous a été fourni, proposez des modifications permettant d'utiliser directement une représentation binaire pour la variable globale `marked`. On vous propose de donner à la variable `marked` le type `int`. On supposera ici qu'un `int` est encodé sur 2 octets et que l'encodage binaire de la valeur 1 sur un emplacement mémoire de 2 octets est : 00000000 00000001.

Question 7.1 (2 points) : modifiez l'initialisation du champ `unique_identifier` dans la fonction `create_vertex` ;

Question 7.2 (1 point) : modifiez la 1ère ligne de la fonction `dfs`, qui indique que le nœud `v` (paramètre de la fonction) a été traité ;

Question 7.3 (1 point) : modifiez la 7ème ligne de la fonction `dfs`, qui teste si un nœud voisin du nœud `v` (paramètre de la fonction) a déjà été traité.