Examen final de programmation impérative

ÉNSIIE, semestre 1

mercredi 9 janvier 2019

Durée : 1h45.
Tout document papier autorisé. Aucun appareil électronique autorisé.
Ce sujet comporte 4 exercices indépendants, qui peuvent être traités dans l'ordre voulu. Il contient 4 pages.
Le barème est donné à titre indicatif, il est susceptible d'être modifié. Le total est sur 20 points.
Certaines questions, précédées par le symbole ($\star$) sont plus difficiles et pourront être traitées à la fin. Il va de soi que toute réponse devra être justifiée, et que toute fonction devra être commentée.

Exercice 1 : Fonctions simples (6 points)

1. Proposer une procédure dont l'effet est de permuter les valeurs de deux variables de type `int` définies auparavant. (La fonction est définie *avant* `main`. À l'intérieur de `main`, les deux variables sont définies et la fonction appelée. On n'écrira pas la fonction `main`.)
2. Soit le type d'enregistrement suivant :

   ```
   struct couple {
     double first;
     double second;
   };
   ```

   Proposer une procédure dont l'effet est de permuter les valeurs des champs `first` et `second` d'un tel enregistrement.
3. Proposer une fonction qui prend en paramètre un tableau de `int` et sa taille, et qui retourne une copie de ce tableau.
4. Proposer une fonction qui prend en paramètre une chaîne de caractère, et qui retourne une copie de cette chaîne.

Exercice 2 : Représentation mémoire (5 points)

On considère les définitions de types suivantes :

```
struct bucket {
  double val;
  struct bucket* next;
};

typedef struct bucket* list;

struct container {
  int key;
  list path;
  struct container *children;
};

typedef struct container* cont;
```

ainsi que le code suivant :

```
  cont a;
  list b;
  struct bucket c;
  struct container d;
  b = malloc(sizeof(struct bucket));
  a = malloc(sizeof(struct container));
  c.val = 3.14;
  b->val = 2.11;
  b->next = &c;
  c.next= calloc(2,sizeof(struct bucket));
  c.next->val = 1.2;
  b->next->next[1].val = 9.11;
  b->next->next[1].next = NULL;
  c.next[0].next = b;
  a->key = 42;
  a->path = b;
  a->children = &d;
  d.key = 24;
  d.path = &c;
  d.children = NULL;
```

1. Représenter l'état de la mémoire à la fin de ces instructions.

On rajoute maintenant à la suite les instructions :

```
free(b);
b = &c;
c.next = NULL;
```

2. Représenter l'état de la mémoire à la fin de ces instructions.
3. ($\star$) Décrire au moins deux problèmes qui apparaissent.

Exercice 3 : Modularité (4 points)

Un projet est découpé en quatre modules : `Server`, `Client`, `Graphics` et `Common`. À la fois `Server` et `Client` font appel à des fonctions de `Common`, tandis que seul `Client` fait appel à des fonctions de `Graphics`. Les autres appels de fonctions sont des appels à des fonctions de son propre module ou de la bibliothèque standard.

On obtient le programme `client` en combinant les modules `Client`, `Graphics` et `Common`, et `server` en combinant `Server` et `Common`.

1. Quelles sont les relations de dépendances entre modules ?
2. Sans `Makefile`, et en supposant que les dépendances sont à jour, quelle commande faut-il écrire dans le shell pour compiler le module `Server` ?
3. Toujours sans `Makefile`, et en supposant que tous les modules nécessaires ont été compilés, quelle commande faut-il taper dans le shell pour créer le programme `server` ?
4. On modifie le module `Graphics` sans changer son interface, quel(s) module(s) faut-il recompiler ?
5. On modifie l'interface du module `Common`, quel(s) module(s) faut-il recompiler ?
6. Écrire le `Makefile` correspondant au projet. (Noter qu'il faut deux cibles pour les exécutables `client` et `server`.)

Exercice 4 : Étude de fonctions (5 points)

On prendra le soin de bien réfléchir avant de donner sa réponse.

1. Soit la fonction `gaussian` suivante :

   ```
   #include <math.h>

   double gaussian(int x, int n) {
     double res;
     res = pow(2, -x * x / n);
     return res;
   }
   ```

   Quelle est la valeur retournée par l'appel `gaussian(1, 2)` ?

2. Soit la procédure `scale` suivante :

```
/*@requires tab est au moins de taille size */
void scale(unsigned char* tab, int size) {
  int i;
  for (i = 0; i < size; i = i + 1) {
    tab[i] = tab[i] * 1.5;
    if (tab[i] > 255)
      tab[i] = 255;
  }
}
```

   Quel effet a cette procédure si on lui passe en argument un tableau contenant `1`, `2`, `10`, `100`, `200` et l'entier `5` ?

3. Soit la fonction `successor_if` suivante :

```
int successor_if(int x, int b) {
  if (b = 1)
    return x + 1;
  else
    return x;
}
```

   Quelle est la valeur retournée par l'appel `successor_if(10, 0)` ?

4. Soit la procédure `successors_else` suivante :

```
void successors_else(int *px, int *py, int b) {
  if (!b)
    *px = *px + 1;
    *py = *py + 1;
}
```

   Quel effet a l'appel `successors_else(&x, &y, 1)` où `x` et `y` sont des variables contenant initialement respectivement les valeurs `42` et `24` ?

5. ($\star$) Soit la fonction `zero_matrix` suivante :

```
#include <stdlib.h>

char **zero_matrix(int n) {
  int i;
  char **res = (char**) malloc(n * sizeof(char));
  for (i = 0; i < n; i++)
    res[i] = (char*) calloc(n, sizeof(char));
  return res;
}
```

   Que se passe-t-il lors d'un appel à `zero_matrix(10)` ?