EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

**Faculté Informatique et Communications**
Cours de Programmation Orientée Système, sections IN et SC
Chappelier J.-C.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

NOM : Hanon Ymous
(000000) #0000
**Place : 0**

# PROGRAMMATION ORIENTÉE SYSTÈME

## Examen

28 mai 2018

## INSTRUCTIONS (à lire attentivement)

**IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre examen annulé dans le cas contraire.**

1. Vous disposez d'une heure quarante-cinq minutes pour faire cet examen (9h15 - 11h00).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur. N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).
3. Vous avez droit à toute documentation papier.
   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée ; ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.
   Vous avez si nécessaire de pages blanches supplémentaires en fin de sujet.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.
6. L'examen comporte trois exercices indépendants, qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 90 points) ; tous les exercices comptent pour la note finale :
   — question 1 : 36 points ;
   — question 2 : 22 points ;
   — question 3 : 32 points.

# Question 1 Questions de cours [36 points]

## Question 1.1 [4 points]

Considérant les déclarations suivantes :

```
int f(int x) { return x + 13; }
int g(int x) { return x * x;  }
int h(int x) { return -2 * x; }

int (*tab1[])(int) = { f, g, h };
int (**p)(int) = tab1;

int tab2[] = { 3, 5, 7, 11 };
int* q = tab2 + 1;
```

que valent chacune des expressions suivantes :

| expression | valeur |
|---|---|
| `(*p)(*q+2)` | |
| `(*p)(*(q+2))` | |
| `(*(p+2))(*q)` | |
| `(*(p+1))(*q)+2` | |

## Question 1.2 [6 points]

Ecrivez à droite ce qu'affiche le code suivant :

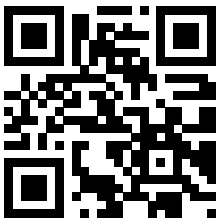
```
char chaine[] = "ABCDEF";
char* p = chaine;
char* q = chaine + strlen(chaine);
while (p++ < --q) {
  *p = *q;
  *q = *(p - 1);
}
puts(chaine);
```

**Affiche :**

Dessinez ensuite (verticalement) la situation en mémoire au départ et à l'arrivée.

**Situation au départ :**

**Situation à l'arrivée :**



## Question 1.3 [6 points]

Ecrivez une fonction `left_pad` qui prend en arguments une chaîne de caractères, un entier positif $n$ ainsi qu'un caractère (exemple ci-dessous) et modifie la chaîne de façon à ce qu'elle contienne $n$ fois le caractère passé en troisième argument suivi du contenu initial (voir exemple ci-dessous). Vous **ne** pouvez utiliser **aucune** fonction de la librarie `string.h` dans votre implémentation (pas même `strlen()` !). Partez par contre du principe que le `char*` passé en argument pointe sur déjà suffisamment de mémoire allouée pour contenir le résultat final.

Par exemple, l'affichage des lignes suivantes serait :

| | affichage |
|---|---|
| `puts(str);` | `Hello world!` |
| `left_pad(str, 2, '#');` | |
| `puts(str);` | `##Hello world!` |

**Définition de la fonction `left_pad()` :**

suite au dos ☞

## Question 1.4 Correction [5 points]

Trouvez les deux erreurs dans le code suivant **et** proposez à chaque fois une correction. Répondez directement sur le code ci-dessous.

```
1   #include <stdio.h>
2   #include <stdint.h> // SIZE_MAX
3
4   size_t find(const int* array, const size_t size, const int element) {
5     if ((array != NULL) && (size != 0)) {
6       for (size_t index = size - 1; index >= 0; --index) {
7         if (array[index] == element) return index;
8       }
9     }
10    return SIZE_MAX;
11  }
12
13  int main()
14  {
15    const int array[] = { 1, 2, 3, 1, 2, 4 };
16    printf("%zu\n", find(array, sizeof(array), 5));
17    return 0;
18  }
```

## Question 1.5 [6 points]

Qu'affiche le programme suivant ? Justifiez *brièvement* votre réponse.

```
#include <stdio.h>

void f1(int* p) {
  (*(p++))++;
  *p = 0;
}

void f2(int** p) {
  (*p)--;
  (**p)--;
}

int main(void) {
  int c[3] = { 1, 2, 3 };
  int* d = c + 1;

  f1(c);
  f1(c + 1);
  f2(&d);
  printf("%d %d %d\n", c[0], c[1], c[2]);
  return 2;
}
```

**Réponse :**

## Question 1.6 [4 points]

Lesquels des programmes suivants compilent sans erreurs ? Justifiez *brièvement* chacune de vos réponses.

```
int main(void) {
  const char* c = "abc";
}
```

```
int main(void) {
  char const* a;
  a = "abc";
}
```

```
int main(void) {
  char* const a;
  a = "abc";
}
```

```
int main(void) {
  const char* a;
  a = "abc";
  a[0] = "b";
}
```

## Question 1.7 [5 points]

Définissez ici une fonction `compact()` qui prend un tableau de pointeurs sur des `int` (possiblement éparpillés dans la mémoire) et retourne un tableau d'`int` (continus en mémoire, donc), ayant le même contenu (mêmes *valeurs* dans le même ordre).

suite au dos ☞

# Question 2 Correction d'erreurs et compréhention de code [22 points]

## Question 2.1 Correction d'erreurs [16 points]

Voici ci-contre un extrait de code C pour implémenter des arbres binaires de recherche. Un tel arbre est défini comme un arbre dont chaque nœud à une valeur et deux sous-arbres (éventuellement vides) tels que la valeur stockée dans ce nœud soit plus petite ou égale à toutes les valeurs stockées dans le sous-arbre de droite et plus grande ou égale à toutes les valeurs stockées dans le sous-arbre de gauche.

Trouvez dans l'extrait de code fourni, un maximum d'erreurs possible **et** proposez à chaque fois une correction. Vous pouvez répondre directement sur le code ci-contre, ainsi qu'à droite de celui-ci ou ci-dessous.

**Attention ! Nous retirons 1 point pour toute indication d'une erreur qui n'en est pas une.** (sinon il suffirait de dire que chaque ligne est fausse...)

**Notes :**

1. On ne vous questionne pas ici sur les arbres binaires de recherche, ce n'est pas le but ; il n'y a donc aucune erreur de théorie à rechercher ; seules des erreurs d'implémentation en langage C nous intéressent ici.
2. Il ne s'agit pas non plus de réécrire tout le code : lorsqu'une erreur potentielle pourrait se corriger de plusieurs façons, choisissez celle qui nécessite le moins de modifications du code fourni.
3. Pour gagner de la place, nous n'avons reproduit qu'une partie du code (c'est pour cela que le code commence à la ligne 4) ; il n'y a pas d'erreur due à du manque de code.

```
 4  typedef struct Node_* BST;
 5
 6  typedef struct Node_ {
 7    int value;
 8    BST* left;
 9    BST* right;
10  } Node;
11
12  int right_most_value(const Node* node) {
13    if (node == NULL) return INT_MIN;
14    if (node->right != NULL) return right_most_value(node->right);
15    else return node->value;
16  }
17
18  int left_most_value(const Node* node); // définie ailleurs
19
20  int is_BST(const BST root) {
21    return root == NULL || (
22        is_BST(root->left ) &&
23        is_BST(root->right) &&
24        (right_most_value(root->left) <= root->value) &&
25        (left_most_value(root->right) >= root->value));
26  }
27
28  Node* insert(BST* root, int value) {
29    if (*root == NULL) {
30      *root = malloc(sizeof(Node));
31      (*root)->left  = NULL;
32      (*root)->right = NULL;
33      (*root)->value = value;
34      return *root;
35    } else if (value < (*root)->value) {
36      return insert((*root)->left,  value);
37    } else if (value > root->value) {
38      return insert((*root)->right, value);
39    }
40  }
41
42  void free_BST(BST root) {
43    free_BST(root->left );
44    free_BST(root->right);
45    free(root);
46    root = NULL;
47  }
```

**suite au dos ☞**

## Question 2.2 Schéma mémoire [6 points]

Une fois le code précédent corrigé, représentez graphiquement l'état de la mémoire correspondant au `BST t` du code suivant :

```
BST t = NULL;
insert(&t,  12);
insert(&t,  34);
insert(&t, -7 );
insert(&t, -68);
insert(&t,  29);
insert(&t,  42);
```

# Question 3 Fonctions d'ordre supérieur et clôtures [32 points]

Dans cette question, nous allons explorer une implémentation en C de ce que l'on appelle les « fonctions d'ordre supérieur » (expliquées maintenant). En C, les pointeurs sur fonction permettent déjà de *stocker* et *transférer* des fonctions, mais pour avoir des « fonctions d'ordre supérieur », il manque encore la notion de clôtures (« *closure* » en anglais), c.-à-d. l'ensemble des variables externes auxquelles cette fonction peut avoir accès (on appelle cela son « environnement lexical »).

Le but de cet exercice sera d'offrir un ensemble d'outils permettant d'écrire du code qui correspondrait au programme *fictif* suivant (lequel utilise une syntaxe imaginaire pour les clôtures) :

```
vector* add_to_vector(vector* input, int y, vector* output) {
    return vector_map(input, int (*lambda)(int x) { return x + y; }, output);
}
```

où « `y` » fait partie de la clôture du pointeur sur fonction `lambda`. Il est évident que la syntaxe employée ci-dessus pour définir la clôture (`int (*lambda)(int x) { return x + y; }`) n'est pas légale en C, mais nous allons justement développer une alternative équivalente.

## Question 3.1 Définition des clôtures [6 points]

Une clôture est donc définie par deux choses : (un pointeur sur) une fonction et un environnement.

Par exemple, pour la fonction

`int f(int x) { return x + y; }`,

la variable `y` est une variable non-locale à la fonction et doit faire partie de son environnement. Par contre, pour la fonction

`int g(int x) { return x - 1; }`,

aucune variable est non-locale et l'environnement de cette fonction sera donc vide. Autre exemple : pour la fonction

`double h(int x) { return a * x + b; }`,

l'environnement sera composé des variables `a` et `b` (de type `double` ici).

Le contenu de l'environnement est donc inconnu *a priori* et sera déterminé lors de la création de la clôture par celui qui la construit (exactement comme l'utilisateur de la fonction `qsort()` fournit à celle-ci les éléments corrects qu'il lui faut lors de son appel). A ce stade, il faut donc utiliser un pointeur générique pour décrire l'environnement. Ce pointeur pointera sur la première variable de l'environnement correspondant (ou sera `NULL` si l'environnement est vide).

De plus, il est clair que du point de vue du C, les (pointeurs sur) fonctions faisant partie de clôtures ne sont pas définies comme ci-dessus, car il *faut* leur passer effectivement tous les arguments. Les vrais pointeurs sur fonctions utilisés dans les trois exemples ci-dessus seront donc en fait :

suite au dos ☞

`int (*p_f)(int x, void* p_y);`

qui pointerait sur une fonction intermédiaire permettant de faire appel à une fonction `f()` telle que :

`int f(int x, int y) { return x + y; };`

de même :

`int (*p_g)(int x, void* unused);`,

pointerait sur une fonction intermédiaire permettant de faire appel à une fonction telle que `g()` donnée plus haut ; et

`double (*p_h)(int x, void* p_a_puis_b);`

pointerait sur une fonction intermédiaire permettant de faire appel à une fonction `h()` telle que :

`double h(int x, double a, double b) { return a * x + b; }`.

Pour simplifier, nous allons dans cette sous-question 3.1 nous concentrer sur les clôtures de fonctions qui prennent un `int` en argument et retournent un `double`. Définissez maintenant la structure `int2double_closure` qui correspond à une clôture de fonction `int` vers `double`, c.-à-d. un pointeur sur une fonction permettant de faire une telle clôture (comme expliqué ci-dessus) et un pointeur (générique) vers son environnement.

Définition de `int2double_closure` :

## Question 3.2 Construction d'une clôture [7 points]

### Question 3.2.1 Fonction outil d'appel intermédiaire [5 points]

Considérons la fonction `h()` définie précédemment :

`double h(int x, double a, double b) { return a * x + b; }`.

Définissez ici une fonction `i_h()` qui prend comme arguments

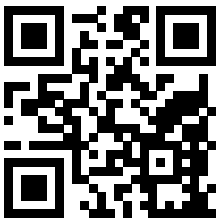
- un entier
- et un tableau de `double` (*sans* sa taille, pour une fois), passé en tant que `void*`,

et retourne un `double` en faisant appel à `h` de telle sorte que `a` soit le premier élément du tableau reçu et `b` le second. Par exemple :

```
double param[] = { 4.3, 5.6 };
int x = 7;

double z = i_h(x, param);
```

(`z` aura alors la valeur de `h(x, 4.3, 5.6)`). Bien sûr `i_h()` doit faire l'hypothèse que le tableau reçu en argument possède au moins deux éléments. On supposera que tel est le cas.



Définition de `i_h()` :

### Question 3.2.2 Construction d'une clôture pour `h` [2 points]

A l'aide de la fonction `i_h()` précédente, définissez une clôture `h_clo` de type `int2double_closure` pour la fonction `h()` et l'environnement donné par le tableau `env` de deux `double` suivant :

```
double env[] = { 1.2, 7.8 };
```

Définition de `h_clo` :

## Question 3.3 Application d'une clôture [4 points]

Maintenant que la structure des clôtures est définie, il faut permettre leur utilisation. Définissez pour cela la fonction `apply_int2double()` qui prend une clôture telle que définie ci-dessus en argument ainsi qu'un entier et applique la clôture à ce dernier. Par exemple, l'appel

```
apply_int2double(h_clo, x)
```

retournera la valeur de `h(x, 1.2, 7.8)`.

Définition de `apply_int2double()` :

suite au dos ☞

## Question 3.4 Opérations sur les tableaux dynamiques [15 points]

Considérez maintenant que vous avez aussi à votre disposition le type `int2void_closure` et la fonction `apply_int2void()` définis de façon similaire à `int2double_closure` et `apply_int2double()` précédents. Considérez de plus avoir à votre disposition un type de tableaux dynamiques `vector`, défini comme suit :

```
typedef struct {
  size_t size;       // number of elements actually stored
  size_t allocated;  // allocated size (in bytes)
  int* content;      // content: array of elements (ints)
} vector;
```

ainsi que les fonctions suivantes (lesquelles sont robustes et ne font rien lorsqu'on leur passe un pointeur `NULL`) :

- `void vector_init(vector* v)` qui initialise le tableau dynamique `v` de manière à ce qu'il corresponde à un tableau vide ;
  il est légal d'appeler `vector_init` sur un tableau déjà initialisé sans libérer son contenu auparavant (ré-initialisation) ;
- `void vector_push(vector* v, int element)` qui ajoute `element` à la fin du tableau `v`.

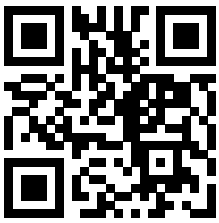
### Question 3.4.1 Définition de la fonction `vector_foreach()` [5 points]

Définissez ici la fonction

`void vector_foreach(const vector* v, int2void_closure f)`

qui parcourt le tableau `v` du début à la fin et applique `f` à chaque élément de `v` (le contenu de `v` lui-même ne peut pas être modifié directement par `f`).

Définition de `vector_foreach()` :



### Question 3.4.2 Fonction `vector_map()` [10 points]

Considérez maintenant que vous avez aussi à votre disposition le type `int2int_closure` et la fonction `apply_int2int()` définis de façon similaire à `int2double_closure` et `apply_int2double()` précédents.

Définissez la fonction

```
vector* vector_map(vector* input, int2int_closure f, vector* output)
```

qui commence par (ré)initialiser le tableau `output` à un tableau vide, puis se sert ensuite de `vector_foreach()` de manière à ce qu'au final le tableau `output` ait la même taille que le tableau `input` et de sorte à ce que pour chaque élément `e` dans le tableau `input`, la valeur à l'index correspondant dans le tableau `output` soit le résultat de l'application de la clôture `f` à l'élément `e`. La fonction `vector_map()` retourne ensuite le tableau `output` ainsi transformé.

**Attention :** pour interagir avec les tableaux dynamiques `vector`, vous ne pouvez utiliser ici *que* les fonctions suivantes : `vector_init()`, `vector_push()` et `vector_foreach()`, et pas autre chose (pas d'accès direct aux champs du `vector`).

**Conseil :** modularisez en définissant des fonctions ou structures auxiliaires pour résoudre cette question.