ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
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.

Programmation Orientée Système (IN/SC) :

CORRIGÉ DE LA
SÉRIE NOTÉE

24 avril 2023

INSTRUCTIONS (à lire attentivement)

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

1. Vous disposez de 1h45 pour faire cette série notée (8h15 - 10h00).
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 ; utilisez aussi le verso des feuilles, **MAIS** n'utilisez *que* le verso de la feuille sur laquelle se trouve la question, et non **pas** celui de la feuille précédente !
   Ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.
   Si nécessaire, il y a une page supplémentaire en fin de copie.
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.
   Si la nature (par valeur ou par référence) d'un passage d'argument n'est pas précisée, c'est *à vous* de faire le choix adéquat.
   Si un comportement ou une situation donnée n'est pas définie dans la consigne, vous êtes libre de définir le comportement adéquat. On considérera comme comportement adéquat toute solution qui ne viole pas les contraintes données et qui ne résulte pas en un crash du programme.
6. Cette série notée ne comporte qu'un seul exercice en neuf questions.

1 – Joli jardin [sur 60 points]

Le but de cet exercice est d'écrire des parties d'un programme permettant de créer de jolis jardins de fleurs (p.ex. pour un jeu). L'idée ultime, non abordée ici, serait la simulation de l'évolution du jardin par croissance de nouvelles fleurs et perte de certaines (p.ex. par manque d'eau). Nous ne nous concentrerons ici que sur certaines parties d'un tel programme.

1.1 – Types de données [sur 8 points]

Les deux types principaux d'un tel programme sont les fleurs (type `Flower`) et le jardin (type `Garden`).

Une fleur contient une couleur (chaîne de caractères de longueur non connue *a priori*), un moyen d'indiquer si elle est nouvelle ou non, et trois compteurs (entiers) : la quantité d'eau requise, la quantité d'eau actuelle et le nombre de jours de sécheresse.

[3 points] Définissez ici le type `Flower` :

```
typedef struct {
    char* color;
    unsigned int water_required;
    unsigned int current_water_level;
    unsigned int dry_days;
    int is_new;
} Flower;
```

Un jardin représente un rectangle de fleurs. Au niveau implémentation, il sera constitué de deux dimension (largeur et longueur du rectangle) et d'un tableau *unidimensionnel* de pointeurs sur des fleurs : en mémoire tous les pointeurs sur les fleurs du jardin sont stockés dans un tableau à *une seule* dimension. **Important :** ce n'est pas à vous à vous préoccuper de comment cette représentation interne unidimensionnelle est faite. Vous supposerez avoir à disposition un moyen d'accéder au pointeur situé en `(i,j)` du rectangle : `flower_at(p_garden, i, j)`, où `p_garden` est un pointeur sur un `Garden`. Vous pourrez (plus tard) utiliser ce moyen aussi bien en lecture qu'en écriture (par exemple : `flower_at(p_garden, i, j) = new_flower(....);`).

[3 points] Définissez ici le type `Garden` :

```
typedef struct {
    size_t x_dim;
    size_t y_dim;
    Flower** flowers; // dynamic array of dynamically allocated flowers
} Garden;
```

[2 points] Définissez ici le tableau `COLORS` (qui ne sera pas changé) des couleurs de base (chaînes de caractères ; qui ne seront pas changées) `"red"`, `"blue"`, `"white"` et `"yellow"` :

```
const char* const COLORS[] = {"red", "blue", "white", "yellow" };
```

1.2 – Fonction outil [sur 6 points]

Comme nous allons devoir créer plein de nouvelles couleurs de fleurs (qui seront composées des couleurs de base, on expliquera plus tard comment), il est utile de définir une fonction `new_substring()` qui retourne une *copie* de la sous-chaîne initiale (préfixe) d'une chaîne reçue en premier paramètre. Le second paramètre de cette fonction est la longueur du préfixe voulu.

Si la longueur demandée pour le préfixe est trop longue (plus longue que la chaîne reçue), alors on retourne `NULL`.

Par exemple, l'appel `new_substring("Bonjour tout le monde !", 7)` retourne la *nouvelle* chaîne `"Bonjour"`, l'appel `new_substring("Bonjour", 10)` retourne `NULL`, et l'appel `new_substring(&s[8], 3)` avec `s = "Bonjour tout le monde !"`, retourne `"tou"`.

Un autre exemple, que vous pourrez aussi utiliser par la suite, est une autre fonction outil qui crée une nouvelle copie d'une chaîne existante :

```
char* new_string_copy(const char* str)
{
    return new_substring(str, strlen(str));
}
```

[6 points] Définissez ici la fonction `new_substring()` :

```
char* new_substring(const char* string, size_t len)
{
    if ((string == NULL) || (len > strlen(string)))  return NULL;

    char* new_str = calloc(len + 1, 1);
    if (new_str == NULL) return NULL;
    strncpy(new_str, string, len);
    return new_str;
}
```

1.3 – Initialisation du jardin [sur 4 points]

Définissez une fonction `prepare_garden()` qui reçoit un `Garden` dont on supposera les tailles (largeur et longueur) déjà affectées, et qui alloue la place nécessaire (largeur fois longueur) pour y stocker les pointeurs sur les fleurs, puis les affecte tous à `NULL`.

Pour rappel, la représentation interne d'un `Garden` est un tableau *unidimensionnel* de pointeurs sur des fleurs, mais ce n'est pas à vous à vous préoccuper de comment cette représentation interne unidimensionnelle est faite. Vous avez à disposition un moyen d'accéder au pointeur situé en `(i,j)` du rectangle : `flower_at(p_garden, i, j)`, où `p_garden` est un pointeur sur un `Garden`. Vous pourrez utiliser ce moyen aussi bien en lecture qu'en écriture (p.ex. `flower_at(p_garden, i, j) = new_flower(....);`). Ces index `(i,j)` vont de 0 à largeur moins un, respectivement : longueur moins 1.

La fonction `prepare_garden()` retourne 0 en cas d'erreur et 1 si tout s'est bien passé.

[4 points] Définition de `prepare_garden()` :

```
int prepare_garden(Garden* garden)
{
    if (garden->x_dim > SIZE_MAX / garden->y_dim) return 0;

    garden->flowers = calloc(garden->x_dim * garden->y_dim,
                             sizeof(Flower*));
    if (garden->flowers == NULL) return 0;

    return 1;
}
```

1.4 – Création de nouvelles fleurs [sur 4.5 points]

Définissez une fonction `new_flower()` qui reçoit une couleur (chaîne de caractères) comme argument et qui retourne un pointeur sur une nouvelle fleur (ou `NULL` en cas de problème), dont la couleur est une *copie* de la couleur reçue.

On pourrait par exemple faire l'appel `flower_at(p_garden, i, j) = new_flower(COLORS[0]);`, ou encore `flower_at(p_garden, i, j) = new_flower("red");`.

Cette fleur est évidemment nouvelle. Et les autres champs seront à 0.

[4.5 points] Définition de `new_flower()` :

```
Flower* create_new_flower(const char* color)
{
    Flower* new_flower = malloc(sizeof(Flower));
    if (new_flower == NULL) return NULL;
    new_flower->color = new_string_copy(color);

    new_flower->is_new = 1;

    // or calloc() above:
    new_flower->water_required = 0;
    new_flower->current_water_level = 0;
    new_flower->dry_days = 0;

    return new_flower;
}
```

1.5 – Libération des fleurs [sur 6.5 points]

Les fleurs étant allouées dynamiquement, il faut bien sûr les libérer. Nous vous demandons pour cela de faire deux fonctions :

- `free_flower()` qui reçoit un pointeur sur une fleur et la libère (ainsi que son contenu si nécessaire) ;
- `remove_flower_at()` qui reçoit un `Garden` et deux coordonnées `i` et `j` et qui, si nécessaire, libère la fleur située en `(i,j)`, puis affecte le pointeur de cette case à `NULL`.
  Rappel : pour accéder aux fleurs d'un jardin via leur coordonnées, vous avez l'outil `flower_at(p_garden, i, j)`.

[2 points] Définissez ici la fonction `free_flower()` :

```
void free_flower(Flower* f)
{
    if (f == NULL) return; // optional
    free(f->color);
    free(f);
}
```

[4.5 points] Définissez ici la fonction `remove_flower_at()` :

```
void remove_flower_at(Garden* garden, size_t i, size_t j)
{
    if (garden == NULL) return; // optional
    if (flower_at(garden, i, j) != NULL) {
        free_flower(flower_at(garden, i, j));
        flower_at(garden, i, j) = NULL;
    }
}
```

1.6 – Affichage compact des fleurs [sur 6.5 points]

Pour pouvoir afficher de façon uniforme le jardin, on souhaite pouvoir créer une représentation des fleurs qui soit une chaîne de toujours 3 caractères. On vous demande pour cela de créer une fonction `flower_to_string()` qui reçoit un pointeur sur une fleur et retourne une (nouvelle) chaîne de caractères qui la représente comme ceci (voir des exemples ci-dessous) :

- le premier caractère est simplement la majuscule (fonction `toupper()`) du premier caractère de la couleur de la fleur ;
- le second caractère est :
  - si la couleur de la fleur contient le caractère `DELIMITER` (défini globalement par ailleurs), alors c'est la majuscule du caractère qui suit le caractère `DELIMITER` ;
  - si la couleur de la fleur **ne** contient **pas** le caractère `DELIMITER`, alors c'est simplement le caractère `DELIMITER` lui-même ;
- le troisième caractère est toujours le caractère `BORDER` (défini globalement par ailleurs).

Par exemple, avec `BORDER = '|'` et `DELIMITER = '-'`,

- si la couleur de la fleur est `"red"`, `flower_to_string()` retourne `"R-|"` ;
- si la couleur de la fleur est `"red-yellow"`, `flower_to_string()` retourne `"RY|"` (sans tiret) ;
- si la couleur de la fleur est `"yellow-blue"`, `flower_to_string()` retourne `"YB|"` (sans tiret).

Note : les couleurs des fleurs contiennent *au plus* une seule fois le caractère `DELIMITER`.

[6.5 points] Définissez ici la fonction `flower_to_string()` :

```
char* flower_to_string(const Flower* f)
{
    char* base = calloc(4, 1);

    base[0] = (char) toupper(f->color[0]);
    base[2] = BORDER;

    const char* const suffix = strchr(f->color, DELIMITER);
    if (suffix != NULL) {
        base[1] = (char) toupper(suffix[1]);
    } else {
        base[1] = DELIMITER;
    }

    return base;
}
```

1.7 – Choix d'une nouvelle sous-couleur [sur 6.5 points]

Nous allons maintenant nous intéresser à des sous-parties du processus de vie des fleurs. Pour cela on aimerait pouvoir créer de nouvelles fleurs ayant (éventuellement) des couleurs composées de leur voisines ; p.ex. une fleur de couleur `"red-yellow"` au milieu de fleurs `"red"` et de fleurs `"yellow"`, ou encore une fleur `"blue-red"` au milieu de fleurs `"yellow-blue"` et de fleurs `"white-red"`.

Nous avons pour cela tout d'abord besoin d'une fonction outil `get_color_contribution()` qui, à partir d'une `Flower` retourne une *nouvelle* couleur (chaîne de caractères) constituée de la façon suivante :

- si la couleur de la fleur reçue ne contient pas le caractère `DELIMITER`, c'est simplement une copie de cette couleur ;
- si, par contre, la couleur de la fleur reçue contient le caractère `DELIMITER`, alors si l'appel à une fonction existante `head_or_tail()` vaut `0` ce sera la couleur *devant* le `DELIMITER` (et sans lui), et sinon, ce sera la couleur *après* le `DELIMITER`.
  Par exemple, si la couleur de la fleur reçue est `"yellow-blue"` et que `head_or_tail()` a retourné `0`, alors `get_color_contribution()` retournera `"yellow"` ; si par contre `head_or_tail()` n'a pas retourné `0`, `get_color_contribution()` retournera `"blue"` (pour cette même fleur).

Note : les couleurs des fleurs contiennent *au plus* une seule fois le caractère `DELIMITER`.

[6.5 points] Définissez ici la fonction `get_color_contribution()` :

```
char* get_color_contribution(const Flower* f)
{
    const char* const suffix = strchr(f->color, DELIMITER);
    if (suffix == NULL) {
        return new_string_copy(f->color);
    } else {
        if (head_or_tail() == 0) {
            return new_substring(f->color, (size_t) (suffix - f->color));
                        // for them: strlen(f->color) - strlen(suffix)
        }

        return new_string_copy(suffix + 1); // for them: &suffix[1]
    }
}
```

1.8 – Nouvelle couleur [sur 10 points]

Il s'agit maintenant de créer une nouvelle couleur à partir de deux fleurs. Écrivez pour cela une fonction `get_new_color()` qui, à partir de deux fleurs reçues en argument, retourne une *nouvelle* couleur (chaîne de caractères) composée de la façon suivante :

- on commence par choisir la contribution de chaque fleur au moyen de la fonction `get_color_contribution()` précédente ;
- si ces deux sous-couleurs sont identiques, on retourne simplement l'une des deux ;
- si elles sont différentes, on retourne la concaténation [1] de la première couleur, du caractère `DELIMITER`, puis de la seconde couleur.

Note : il faudra bien faire attention à ne pas avoir de fuite de mémoire.

[10 points] Définissez ici la fonction `get_new_color()` :

```
char* get_new_color(const Flower* f1, const Flower* f2)
{
    char* color1 = get_color_contribution(f1);
    char* color2 = get_color_contribution(f2);

    char* new_color = NULL;
    if (strcmp(color1, color2)) {
        // in case color2 != color1: concatenate color1 and color2
        const size_t sl1 = strlen(color1);
        // this could also be done via a new alloc + free(color1)
        new_color = realloc(color1, sl1 + 1 + strlen(color2) + 1);
        if (new_color != NULL) {
            color1 = new_color;
            color1[sl1] = DELIMITER;
            strcpy(color1 + sl1 + 1, color2);
            /* or for them:
             *   color1[sl1+1] = '\0';
             *   strcat(color1, color2);
             */
        }
    }
    // otherwise (color2 == color1): simply return color1

    free(color2);
    return color1;
}
```

1. c.-à-d. la mise bout à bout

1.9 – Choix d'un voisin [sur 8 points]

Pour finir, nous souhaiterions pouvoir trouver une fleur voisine d'une case donnée du jardin, qui n'ait pas la même couleur et qui ne soit pas une nouvelle fleur.

Définissez pour cela une fonction `get_different_neighbour()` qui prend en paramètre un `Garden` et deux coordonnées `i` et `j`, et qui, parmi les quatre (ou moins) fleurs voisines situées en `(i-1, j)`, `(i+1, j)`, `(i, j-1)`, `(i, j+1)`, lorsqu'il s'agit bien d'une position dans le jardin, retourne la première fleur qui ne soit pas une nouvelle fleur [2] et qui n'a pas la même couleur que celle en `(i,j)`.

La fonction `get_different_neighbour()` retourne `NULL` si elle ne trouve pas une telle fleur.

Indication : (comme toujours,) évitez tout « copié-collé ».

[8 points] Définissez ici (ou sur la page en face) la fonction `get_different_neighbour()` :

```
Flower* get_different_neighbour(const Garden* garden, size_t i, size_t j)
{
    if (flower_at(garden, i, j) == NULL) return NULL;
#define NEIGHBOURS 4
    const size_t is[NEIGHBOURS] = { i-1, i+1,  i ,  i };
    const size_t js[NEIGHBOURS] = {  j ,  j , j-1, j+1};

    for (size_t k = 0; k < NEIGHBOURS; k++) {
        const size_t nF_i = is[k];
        const size_t nF_j = js[k];
        if (nF_i < garden->x_dim && nF_j < garden->y_dim) {
            Flower* neighbour = flower_at(garden, nF_i, nF_j);

            // Third condition: do not allow mutation with a flower that was created today
            if (neighbour != NULL && strcmp(neighbour->color, flower_at(garden, i, j)->color)
                && neighbour->is_new == 0) {
                return neighbour;
            }
        }
    }
    return NULL;
}
```

2. Revoir si nécessaire en 1.1 les attributs d'une fleur.