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.

NOM : Hanon Ymous
(000000)
Place : 0

#0000

PROGRAMMATION ORIENTÉE SYSTÈME

Examen

26 juin 2021

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 de trois heures pour faire cet examen (16h15 – 19h15).
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**.
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 quatre exercices indépendants (sur 16 pages), qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose :
   — question 1 : 25 points ;
   — question 2 : 20 points ;
   — question 3 : 30 points ;
   — question 4 : 45 points ;

   le total est de 120 points ; tous les exercices comptent pour la note finale.

REMARQUE : pour éviter toute ambiguïté de terme, nous désignerons par « tableau » (entre guillemets) un tableau au sens algorithmique du terme (on aurait aussi pu dire « liste de valeurs »), et par tableau C (sans guillemet) ou « array », un tableau au sens du C (comme par exemple : `double t[12];`). Vous n'êtes donc pas du tout obligé(e)s d'implémenter un « tableau » (entre guillemets) par un tableau C (« array »).

Question 1 – Quelques questions [25 points]

1.1 Dessine moi une mémoire [5 points]

Avec les déclarations données à droite, que sont/représentent/valent `ptr1 + 7` et `ptr2 + 7` ?
Dessinez une image de la mémoire et indiquez y les 4 variables et les 2 réponses.

```
double*  tab1[6];
double   tab2[5][4];
double** ptr1 = tab1;
double*  ptr2 = tab2;
```

Réponse :

1.2 Algèbre [4 points]

Que pensez-vous du code suivant ?
Étayez/Justifiez votre réponse.

```
typedef double* vector;

vector subtract(const vector v1,
                const vector v2,
                size_t size)
{
  vector result;
  for (size_t i = 0; i < size; ++i) {
    result[i] = v1[i] - v2[i];
  }
  return result;
}
```

Réponse :

(ne pas écrire dans cette zone)

1.3 C vit ! [3.5 points]

Ce programme compile-t-il et s'exécute-t-il correctement ? Si oui, qu'affiche-t-il ? Dans tous les cas, justifiez votre réponse.

Réponse :

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

void f(int* q) {
  q = malloc(sizeof(int));
  *q = 45;
}

int main(void)
{
  int  i   = 3;
  int* ptr = &i;
  f(ptr);
  printf("%d %d\n", i, *ptr);
  return 0;
}
```

1.4 Hélène de Troie [4.5 points]

Ce programme compile-t-il et s'exécute-t-il correctement ? Si oui, qu'affiche-t-il ? Dans tous les cas, justifiez votre réponse.

Réponse :

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

double* f(double* q) {
  q = malloc(sizeof(double));
  *q = log(3.0); // 1.098612
  return q;
}

int main(void)
{
  double x  = 9.8;
  double* ptr1 = &x;
  double* ptr2 = f(ptr1);
  printf("%f\n", *ptr2);
  free(ptr1);
  return 0;
}
```

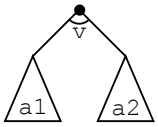
suite au dos ☞

1.5 Branchés [4 points]

On s'intéresse à des arbres binaires, dont les nœuds ont une valeur et deux sous-arbres :

```
typedef struct Noeud_ {
  Type valeur;
  Arbre* droit;
  Arbre* gauche;
} Noeud;
```

Écrivez ci-dessous la définition de la fonction `join()` qui, étant donnés une valeur `v` et deux (sous-)arbres *existants* `a1` et `a2`, passés en paramètre, retourne le `Noeud` qui les relie :



Réponse :

1.6 Ras C [4 points]

Ce programme compile-t-il et s'exécute-t-il correctement ? Si oui, qu'affiche-t-il ? Dans tous les cas, justifiez votre réponse.

```
#include <stdio.h>

void f(char tab[6]) {
    *(tab++ + 2) = '\0';
    *(tab + 5) = '\0';
    printf("%s\n", tab + 3);
}

int main(void)
{
  char tab[] = "Ce chat de race";
  f(tab);
  return 0;
}
```

Réponse :

(ne pas écrire dans cette zone)

Question 2 – Saisie d'une suite de valeurs [20 points]

Le but de cet exercice est d'écrire une fonction `read_vector(void)` qui retourne un « tableau » de valeurs (décimales) entrées par l'utilisateur, mais dont on ne connaît pas *a priori* la longueur. C'est l'utilisateur qui décide combien de valeurs entrer et *termine* la saisie en entrant une lettre.

Par exemple, dans l'interaction suivante où l'utilisateur a décidé d'entrer quatre valeurs :

```
Entrez une suite de valeurs :
1.1 2.2 3.3
4.4
q
```

la fonction `read_vector()` retournera un « tableau » contenant, dans cet ordre, les valeurs 1.1, 2.2, 3.3 et 4.4.

Dans cet autre exemple :

```
Entrez une suite de valeurs :
9.5 z
```

la fonction `read_vector()` retournera un « tableau » contenant uniquement la valeur 9.5.

Si l'utilisateur entre tout de suite une lettre (donc aucune valeur), la fonction `read_vector()` retournera un « tableau » vide.

2.1 Type de retour [4 points]

Définissez ici (et expliquez) le type de retour que vous proposez pour la fonction `read_vector(void)`.

Réponse :

2.2 Définition [16 points]

Définissez ici (et sur la page suivante) la fonction `read_vector()`. Vous pouvez, si vous le souhaitez, définir des fonctions-outils supplémentaires.

Réponse :

Suite de la réponse à 2.2 :

Question 3 – Pluriels de mots [30 points]

Le but de cet exercice est d'écrire *quelques* « fonctions-outils » d'un programme dont le but serait de générer les pluriels d'une liste de mots (« dictionnaire »). Pour simplifier, nous ne considérons ici que les pluriels consistant à ajouter un 's' final (p.ex. chat $\longrightarrow$ chats) ou à transformer un « al » final en « aux » (p.ex. animal $\longrightarrow$ animaux) *sans* considérer les exceptions (bal, carnaval, chacal, ...).

À noter que vous pouvez écrire certaines des fonctions demandées sans nécessairement être parvenu à écrire celles qui précèdent, mais en supposant simplement qu'elles existent.

3.1 Type de données [3 points]

La première chose dont nous avons besoin est de représenter des « dictionnaires », liste de mots, dont on ne connaît pas forcément la taille *a priori*.
Définissez ici (et expliquez très brièvement) le type que vous proposez pour les « dictionnaires ».

Réponse :

3.2 Affichage [4 points]

Afin que nous comprenions mieux l'utilisation de votre structure de données, définissez ici une fonction `affiche()` qui reçoit un « dictionnaire » en paramètre et l'affiche, un mot par ligne.

Réponse :

(ne pas écrire dans cette zone)

suite au dos ☞

3.3 Initialisation [4 points]

Définissez ici la fonction `initialize()` qui reçoit un « dictionnaire » et une taille et qui initialise ce « dictionnaire », comme vide si la taille est 0, et ayant la place nécessaire à recevoir n « mots » si la taille passée est n.

Réponse :

3.4 Suffixes [6 points]

Pour pouvoir déterminer quelle règle de pluriel nous devons appliquer, nous avons besoin d'une fonction déterminant si une chaîne donnée termine par une autre ou non.

Définissez ci-dessous la fonction `has_suffix()` qui reçoit deux chaînes de caractères et retourne 0 si la seconde *n'*est *pas* suffixe de la première, et une valeur non nulle si elle l'est.

Par exemple, « `has_suffix("animal", "al")` » retournera une valeur non nulle,
et « `has_suffix("chien", "al")` » retournera 0.

À noter qu'une chaîne suffixe est forcément plus courte (ou de même longueur) que la chaîne dont elle est le suffixe.

Réponse :

(ne pas écrire dans cette zone)

3.5 Pluriel de mot [9 points]

Définissez ici la fonction `plural_word()` qui, étant donnée une chaîne de caractères, retourne la *nouvelle* chaîne correspondant à son pluriel.

Pour simplifier, nous ne considérons ici que les pluriels consistant à ajouter un 's' final ou à transformer un « al » final en « aux », *sans* considérer les exceptions (bal, chacal, ...).

Par exemple, « `plural_word("chat")` » retournera `"chats"` et « `plural_word("animal")` » retournera `"animaux"`.

Réponse :

(ne pas écrire dans cette zone)

suite au dos ☞

3.6 Pluriel de dictionnaire [4 points]

Définissez ici la fonction `plural_dictionary()` qui, étant donné un « dictionnaire », retourne un *nouveau* « dictionnaire » de même taille, mais contenant les pluriels des mots du « dictionnaire » initial.

Par exemple, si le « dictionnaire » reçu contient les mots : « animal », « chien », « chat » et « canal » (dans cet ordre), alors `plural_dictionary()` retournera le « dictionnaire » contenant (dans cet ordre) les mots « animaux », « chiens », « chats » et « canaux ».

Réponse :

Question 4 – Segmentation d'expressions [45 points]

Le but de cet exercice est de fournir quatre « fonctions-outils » pour un programme de segmentation de chaîne de caractères. Nous présentons d'abord le cadre général, avant d'expliciter ce qui est demandé. La signature et le rôle de chaque fonction demandée sont expliqués dans les sous-sections suivantes.

4.1 Cadre général [0 point]

Il n'y a rien dans cette section que vous ayez à faire ; elle ne fait que présenter le cadre général pour bien comprendre ce qui vous est demandé dans les sections suivantes. *Mais* nous pensons qu'il vaut la peine de prendre le temps (au moins 6 min.) de bien la lire.

4.1.1 Objectif

Le but général visé est de pouvoir représenter plusieurs segmentations (« *tokenization* ») possibles d'une chaîne de caractères, comme par exemple les segmentations « carte bleue » (1 seul segment (« *token* »)) et « carte », « bleue » (2 segments). Un exemple exact de déroulement est donné au dos, en sous-section 4.1.3.

4.1.2 Structure de données

Une segmentation d'une chaîne de caractères sera représentée comme un « tableau » de pointeurs sur cette chaîne. Plutôt que d'indiquer explicitement la taille de ce « tableau », celui-ci contiendra simplement comme dernier élément le pointeur vers le caractère nul en fin de la chaîne segmentée.
Par ailleurs, comme le premier segment commence toujours au début de la chaîne, on **n'**ajoutera **pas** le pointeur sur le début de la chaîne comme premier élément de ce « tableau ».

Par exemple, si la chaîne est : `const char* s = "carte bleue";` ,
la segmentation en 2 mots sera représentée par le « tableau » contenant `s+5` et `s+11`.

Comme l'on souhaite pouvoir avoir un accès en $\mathcal{O}(1)$ à chacune des segmentations d'une chaîne donnée, ces segmentations sont elles-mêmes stockées dans un « tableau » : un « tableau » de « tableaux », donc.

Par exemple, toujours pour la chaîne `s` ci-dessus, les deux segmentations présentées plus haut (« carte bleue » (1 segment) et « carte », « bleue » (2 segments)) seraient représentées dans un « tableau » contenant 2 « tableaux » : le premier « tableau » contenant simplement un seul élément, `s+11`, et le second « tableau » étant celui illustré ci-dessus (`s+5` et `s+11`).

On arrive ainsi à la structure suivante pour représenter les segmentations d'une chaîne (voir son utilisation dans la sous-section suivante) :

```
typedef struct {
  const char* s;
  size_t n;
  const char*** tokens;
} tokenizations;
```

où :
- `s` est la chaîne dont on veut les segmentations ;
- `n` est le nombre de segmentations ;
- `tokens` est le « tableau » de « tableaux » de segments.

suite au dos ☞

4.1.3 Exemple de `main()`

Par exemple, le programme suivant (dont les fonctions utilisées sont l'objet de cet exercice ; pour rappel, la signature et le rôle de chacune de ces fonctions sont détaillés dans les sous-sections suivantes) :

```
int main(void)
{
  tokenizations example = {
    "mise à pied à terre rare",
    0, NULL
  };

  // "mise à pied", "à", "terre", "rare"
  size_t positions[3] = { 11, 13, 19 };
  add_tokenization(&example, 3, positions);

  // "mise à pied", "à", "terre rare"
  add_tokenization(&example, 2, positions); // notez le 2 ici

  // "mise", "à", "pied à terre", "rare"
  positions[0] = 4; positions[1] = 6;
  add_tokenization(&example, 3, positions);

  // tous les mots
  add_whitespace_tokenization(&example);

  print_tokenizations(&example);
  release(&example);
  return 0;
}
```

afficherait :

```
"mise à pied", " à", " terre", " rare",
"mise à pied", " à", " terre rare",
"mise", " à", " pied à terre", " rare",
"mise", " à", " pied", " à", " terre", " rare",
```

Remarques :

1. On supposera que `'à'` n'utilise qu'un seul `char`.
2. Le `main()` ci-dessus est juste un exemple ; il ne construit pas toutes les segmentations possibles en français de cette expression.
3. Pour simplifier, nous avons décidé de ne supprimer aucun caractère : dans l'exemple ci-dessus, tous les segments après le premier commence donc par une espace [1].

Ce qui vous est demandé dans la suite est d'écrire les 4 fonctions : `print_tokenizations()`, `add_tokenization()`, `add_whitespace_tokenization()` et `release()`.
À noter que vous pouvez écrire ces fonctions sans nécessairement être parvenu à écrire les autres, mais en supposant qu'elles existent.

1. L'espace du typographe est féminine.

4.2 Affichage des segmentations [10 points]

On vous demande ici d'écrire la fonction `print_tokenizations()` telle qu'utilisée dans l'exemple précédent (sous-section 4.1.3) et dont le but est d'afficher, ligne à ligne, toutes les segmentations d'une chaîne. Voir l'exemple précédent (sous-section 4.1.3) pour le format d'affichage.

On supposera pour cela l'existence d'une fonction

```
void print_from_to(const char* start, const char* end);
```

qui affiche entre guillemets tous les caractères compris entre `start` (inclus) et `end` (exclu). Par exemple, le code suivant afficherait « `"jour"` » :

```
const char* s = "Bonjour !";
print_from_to(s + 3, s + 7);
```

Écrivez ici la définition de la fonction `print_tokenizations()` :

(ne pas écrire dans cette zone)

suite au dos ☞

4.3 Ajout d'une segmentation [20 points]

On vous demande ici d'écrire la fonction `add_tokenization()` telle qu'utilisée dans l'exemple de la sous-section 4.1.3, et dont le but est d'ajouter une segmentation à l'ensemble des segmentations d'une chaîne. Cette nouvelle segmentation est fournie sous la forme d'un tableau C (« array ») contenant les positions des fins (exclues) des segments désirés. Voir l'exemple de la sous-section 4.1.3 pour son utilisation exacte.

À noter que, même s'il n'est pas utilisé dans le `main()` d'exemple donné en sous-section 4.1.3, vous pouvez néanmoins faire/supposer que cette fonction retourne un code d'erreur (`int`), si cela vous semble utile.

Écrivez ici la définition de la fonction `add_tokenization()` :

4.4 Ajout de la segmentation sur tous les espaces [10 points]

On vous demande ici d'écrire la fonction `add_whitespace_tokenization()` telle qu'utilisée dans l'exemple de la sous-section 4.1.3, et dont le but est d'ajouter la segmentation consistant à créer un segment sur chaque espace rencontrée. On supposera que la chaîne reçue est « bien formée » en ce sens qu'une seule espace sépare chaque segment (et qu'il y a au moins un segment valide).

Et, là aussi, on pourra faire/supposer que cette fonction retourne un code d'erreur, si cela vous semble utile.

Écrivez ici la définition de la fonction `add_whitespace_tokenization()` :

suite au dos ☞

4.5 Garbage collecting [5 points]

On vous demande ici d'écrire la fonction `release()` telle qu'utilisée dans l'exemple de la sous-section 4.1.3, et dont le but est de libérer la mémoire allouée par les segmentations d'une chaîne.

Après un appel à `release()`, on devrait pouvoir relancer une nouvelle segmentation sur la chaîne ; p.ex. lancer à nouveau `add_whitespace_tokenization()` si nécessaire (comme si on repartait du début).

Écrivez ici la définition de la fonction `release()` :