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

Correction 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 18 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 »).

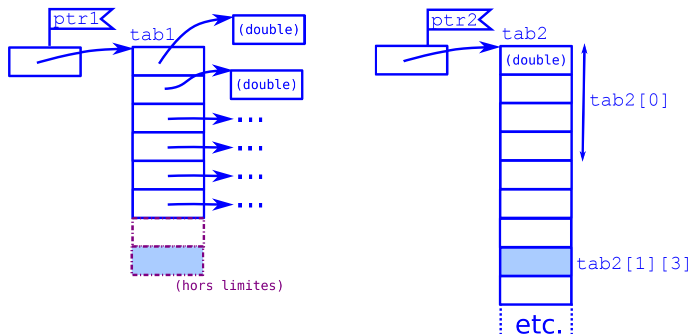
Question 1 – Quelques questions [25 points]

1.1 Dessine moi une mémoire [5 points]

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

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.

Réponse : `ptr1 + 7` pointe vers de la mémoire non autorisée (2 au delà de la fin de `tab1`) et `ptr2 + 7` pointe vers `tab2[1][3]`.



Commentaires de correction : réussie à 64%. Les principales erreurs ont été :

— mauvaises compréhentions de l'organisation mémoire ;
— en particulier un tableau à deux dimensions (`tab2`) est continu en mémoire et ne doit pas être confondu avec un tableau de pointeurs ;
— confusion entre pointeur et contenu pointé ; p.ex. `ptr2+7` n'est pas égal à `tab2[1][3]`, mais à son *adresse* !

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 : 2 critiques :

1. La plus importante : `result` non alloué ; il faudrait `[mc]alloc` à `size` fois `sizeof(double)`.
2. Conception : rien n'assure que la taille passée soit effectivement la taille de `v1` et `v2`, et surtout qu'ils aient la même taille.

   ce serait mieux de les grouper : `struct Vecteur { double* content; size_t taille; };` et de tester l'égalité des tailles

On pourrait aussi (fonction d'interface) s'assurer que `v1` et `v2` ne sont pas `NULL`.

Commentaires de correction : réussie à 67%. Les principales erreurs ont été :

- ne pas dire comment faire l'allocation
- ne pas critiquer la taille
- certains évoquent le retour d'une variable locale sans avoir remarqué (ou compris le rôle ?) que `vector` est justement un pointeur.

En raison du fort taux des deux premières erreurs citées ci-dessus (lié à la formulation trop imprécise de la question), j'ai décidé de réduire mon barème d'autant (=transformer en bonus). Le taux de réussite est alors de 91%.

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.

```
#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;
}
```

Réponse : Oui : `3 3`; `ptr` étant passé par valeur, les modifications faites par `f()` n'ont aucun impact. Il y a par contre une fuite de mémoire !

Commentaires de correction : réussie à 44%. Les principales erreurs ont été :

- ne pas comprendre le passage par *valeur* du pointeur
- ne pas répondre à toutes les questions : s'il y a *deux* points-d'intérrogation, il faut au moins *deux* réponses !

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.

```
#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;
}
```

Réponse : Non : il compile en effet mais ne s'exécute pas correctement et fait un « segmentation fault » après avoir affiché `1.098612`; `ptr` étant passé par valeur, les modifications faites par `f` n'ont aucun impact et on ne libère donc pas la bonne chose : on cherche ici à libérer l'adresse, allouée statiquement, de `x` ; ce qui fait une erreur mémoire (probablement un « double free » en fait). Il y a donc une fuite mémoire sur le `malloc`.

Commentaires de correction : réussie à 19%. Énormément de confusions sur cette question. Les principales erreurs ont été :

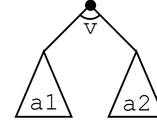
- les deux mêmes que pour 1.3 ;
- confondre compilation et exécution
- ne pas voir le « segmentation fault » (double free)

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 :

Il y a plusieurs façon d'écrire cette fonction. Par exemple :

```
Noeud joint(Type v, Arbre* a1, Arbre* a2)
{
  return (Noeud) { v, a2, a1 };
}
```

Notes :

1. on doit pouvoir autoriser `a1` et `a2` à être `NULL` (cas des feuilles) ;
2. la fonction peut aussi très bien retourner un `Noeud*`. Il devra alors avoir été alloué dynamiquement. Par exemple :

```
Noeud* joint(Type v, Arbre* a1, Arbre* a2)
{
  Noeud* n = malloc(sizeof(Noeud));
  if (n != NULL) {
    n->valeur  = v;
    n->gauche  = a1;
    n->droit   = a2;
  }
  return n;
}
```

Mais en *aucun* cas il ne faut faire l'allocation dynamique des sous-arbres !

Commentaires de correction : réussie à 81%. Les principales erreurs ont été :

- de prendre `Noeud` pour un pointeur ;
- de passer des `const Arbre*` ;
- de confondre `Arbre` et `Noeud*` (rien ne dit que c'est le cas ; on pourrait très bien vouloir ajouter d'autres informations au niveau de l'`Arbre`)

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 :

Oui : `ha` ;

`tab` est de toutes façons passé comme un pointeur, donc le « `[6]` » n'a aucun rôle ;

la première ligne coupe la chaîne entre « Ce » et « chat », puis avance le pointeur `tab`, ce qui fait que la seconde ligne coupe la chaîne après le `'a'` (au 7e caractère : 5e position de « `tab+1` ») ;

enfin on affiche cette chaîne (coupée) à partir de son 5e caractère (3e position de « `tab+1` »).

Commentaires de correction : réussie à 48%. Les principales erreurs ont été :

- tomber dans le « piège » du `tab[6]` : je répète qu'un tableau est toujours passé comme un simple pointeur.
- confusion entre le `tab` fourni et un « `const char* tab = "...";` »
- ne pas répondre aux *deux*/trois questions.

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 :

```
typedef struct {
  size_t  size;
  double* content;
} vector;
```

On peut, bien sûr, plus sophistiquer ; p.ex. en ajoutant une taille allouée (« capacité ») ou avec un « flexible array member ».

Commentaires de correction : réussie à 97%. Les principales erreurs ont été :

— oublier la taille (il est dommage que le « pattern » de « tableau dynamique » ne soit pas encore maîtrisé...)

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 :

Il y a plusieurs solutions possibles. En voici *une*, minimale et à croissance incrémentale, suivie d'une plus complète à croissance exponentielle de la taille allouée :

```
vector read_vector(void)
{
  vector retour;
  retour.size = 0;
  retour.content = malloc(sizeof(double));
  if (retour.content == NULL) return retour;

  puts("Entrez une suite de valeurs :");
  double read = 0.0; // input buffer
  while (scanf("%lf", &read) == 1) {
    retour.content[retour.size++] = read;

    // prepare la place pour le suivant
    double* new = realloc(retour.content, (retour.size + 1) * sizeof(double));
    if (new == NULL) return retour;
    retour.content = new;
  }

  return retour;
}
```

Ce qui est important c'est :

- s'assurer de la bonne lecture (pas besoin de « réinventer la roue » ici, `scanf()` fait bien l'affaire) ;
- de ne pas perdre l'original en cas d'échec du `realloc()`.

Remarques :

- Ce n'est pas « grave » que le vecteur retourné ait alloué un (ou même « quelques ») `double` en plus.
- La fonction pourrait aussi retourner un pointeur sur un `vector` ; lequel aura alors, bien entendu, été alloué dynamiquement.
  (Cela est nécessaire pour les « flexible array member ».)

```
double* realloc_or_free(double* p, size_t s)
{
  double* const new = realloc(p, s);
  if (new == NULL) {
    free(p);
  }
  return new;
}

vector read_vector(void)
{
  vector retour = { 0, NULL };

  size_t size = 32; // buffer initial size (any decent start value)
  retour.content = calloc(size, sizeof(double));
  if (retour.content == NULL) return retour;

  puts("Entrez une suite de valeurs :");
  double read = 0.0; // input buffer

  // pas vraiment attendu des etudiants, un peu « surfait »
  const size_t too_big = SIZE_MAX / (2 * sizeof(double));

  while (!feof(stdin) && !ferror(stdin) // pas vraiment attendu des etudiants non plus

         && (scanf("%lf", &read) == 1)) {

    if (retour.size >= size) {
      if (size > too_big) { // optionnel (bonus)
        free(retour.content);
        retour.content = NULL;
        retour.size = 0;
        return retour;
      }
      size *= 2; // exponentially growing buffer
      retour.content = realloc_or_free(retour.content, size * sizeof(double));
      if (retour.content == NULL) {
        retour.size = 0;
        return retour;
      }
    }
    retour.content[retour.size++] = read;
  }

  // optional downsizing (here no risk of multiplication overflow)
  retour.content = realloc(retour.content, retour.size * sizeof(double));

  return retour;
}
```

Commentaires de correction : réussie à 60%. Les principales erreurs ont été :

- beaucoup trop se compliquer la vie !
- faire de l'allocation de taille fixe (alors qu'il est clairement spécifié qu'on ne connait pas la taille) ;
- réallocation avec `calloc()` au lieu de `realloc()` ;
- ne pas correctement gérer le cas d'échec de `realloc` (perte de l'ancien contenu).

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 ⟶ chats) ou à transformer un « al » final en « aux » (p.ex. animal ⟶ 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 :

```
typedef struct {
  const char** content; // tableau dynamique de chaines de caracteres
  size_t       size;
} Dico;
```

On peut, bien sûr, plus sophistiquer ; p.ex. en ajoutant une taille allouée (« capacité ») ou avec un « flexible array member ».

Commentaires de correction : réussie à 90%. Les principales erreurs ont été :

- comme pour 2.1, oublier la taille ;
- utilisation d'allocation statique (taille fixe) pour les mots
- ne pas mettre le `const`.

En raison de l'aspect finalement discutable de ce `const`, j'ai décidé de réduire mon barème d'autant (c.-à-d. le transformer en bonus). Le taux de réussite est alors de 96%.

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 :

```
void affiche(const Dico* d)
{
  if (d == NULL) return; // optionnel

  if (d->content != NULL) // nécessaire
  {
    for (size_t i = 0; i < d->size; ++i) {
      printf("- \"%s\"\n", d->content[i]); // format libre (sauf \n)
    }
  }
}
```

La `struct Dico` étant petite, on peut tolérer un passage par valeur.

Notez que `printf("%s")` est déjà lui-même protégé contre `NULL`.

Commentaires de correction : réussie à 67%. Les principales erreurs ont été :

- ne pas vérifier `content` avant d'y accéder
- s'inventer des `NULL` qui n'existent pas (confusion avec `'\0'` ?)
- utiliser `%s` pour des `char` (ou trop déréférencer)

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 :

```
void initialize(Dico* d, size_t taille)
{
  if (d == NULL) return; // optionnel
  d->size = taille;
  if (taille != 0) {
    d->content = calloc(taille, sizeof(char*));
    if (d->content == NULL) d->size = 0;
  } else {
    d->content = NULL;
  }
}
```

Commentaires de correction : réussie à 85%. Les principales erreurs ont été :

- ne pas passer le `Dico` par référence (ou le passer par référence constante !)
- ne pas (correctement) gérer le cas de la taille nulle
- confusion `sizeof(char)` au lieu de `sizeof(char*)`

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 :

```
int has_suffix(const char* mot, const char* suffixe)
{
  if (mot     == NULL) return 0; // optionnel
  if (suffixe == NULL) return 0; // optionnel

  const size_t l_mot     = strlen(mot);
  const size_t l_suffixe = strlen(suffixe);

  if (l_mot < l_suffixe) return 0;

  return !strncmp(mot + l_mot - l_suffixe, suffixe, l_suffixe);
}
```

Attention à ne pas pointer avant le mot si le suffixe est plus long que le mot !

Commentaires de correction : réussie à 89%. Les principales erreurs ont été :

- se compliquer la vie (ne pas utiliser les fonctions standard)
- utiliser `strstr()` (qui trouve la *première* occurence de la chaîne)
- ne pas gérer le cas de suffixes (trop) longs
- erreur d'index

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 :

```
char* plural_word(const char* recu)
{
  if (recu == NULL) return NULL; // optionnel

  const size_t taille = strlen(recu);
  char* const retour = calloc(1, taille + 2) ; // le pluriel (+1 char) et le '\0'
  if (retour == NULL) return retour;

  strcpy(retour, recu);
  if (has_suffix(recu, "al")) {
    retour[taille-1] = 'u';
    retour[taille]   = 'x';
  } else {
    retour[taille] = 's';
  }

  // si ci-dessus malloc au lieu de calloc :
  // retour[taille+1] = '\0';

  return retour;
}
```

Commentaires de correction : réussie à 85%. Les principales erreurs ont été :

- `calloc` trop courts (de 1)
- modification/realloc de la chaîne de référence (`s`)
- erreurs d'index

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 :

Voici *une* version (minimale) possible :

```
Dico plural_dictionary(const Dico* d)
{
  Dico retour;
  initialize(&retour, d->size);
  for (size_t i = 0; i < retour.size; ++i) {
    retour.content[i] = plural_word(d->content[i]);
  }
  return retour;
}
```

et une autre :

```
Dico plural_dictionary(const Dico* d)
{
  Dico retour = { .size = 0, .content = NULL };

  if (d == NULL) return retour;
  if (d->content == NULL) return retour;

  initialize(&retour, d->size);

  for (size_t i = 0; i < retour.size; ++i) {
    if (d->content[i] != NULL) { // test optionnel (bonus)
      retour.content[i] = plural_word(d->content[i]);
    }
  }

  return retour;
}
```

Remarques :

- La `struct Dico` étant petite, on peut tolérer un passage par valeur.
- On pourrait aussi retourner un pointeur sur un `Dico` ; lequel serait alors, bien entendu, été alloué dynamiquement.
  (Cela serait nécessaire pour les « flexible array member ».)

Commentaires de correction : réussie à 80%. Les principales erreurs ont été :

- oubli d'allocation
- diverses confusion entre pile (stack) et tas (heap) : p.ex. return de l'adresse d'un struct alloué sur la stack ;
- allocation de mémoire dynamique de la taille du mot retourné par `plural_word` puis `strncpy` dans le nouveau dictionnaire plutôt que simple assignation

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.

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()` :

```
void print_tokenizations(const tokenizations* t)
{
  if (t == NULL) return; // bonus (1 seule fois)

  for (size_t i = 0; i < t->n; ++i) {
    const char** q = t->tokens[i];
    for (const char* p = t->s; *p; p = *q++) {
      print_from_to(p, *q);
      putchar(','); putchar(' '); // ou fputs(", ", stdout) ou printf
    }
    putchar('\n');
  }
}
```

Il n'est pas strictement nécessaire d'utiliser ici l'arithmétique des pointeurs, mais c'est tellement plus élégant...

La `struct tokenization` étant petite, on peut tolérer un passage par valeur.

Commentaires de correction : réussie à 89%. Les principales erreurs ont été :

- utilisation de `strlen()` ou, pire, `sizeof()` pour le parcourt
- un niveau de déréférencement en moins au moment de passer les pointeurs à `print_from_to()`
- confusions entre tokenisations et tokens

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()` :

```
int add_tokenization(tokenizations* t, size_t nb, size_t pos[])
{
  if (t == NULL) return 1; // bonus (1 seule fois)

  const size_t new_n = t->n + 1;
  // bonus: tester que la multiplication n'overflow pas
  const char*** const new_tokens = realloc(t->tokens, new_n * sizeof(void*));
  if (new_tokens == NULL) return 1;

  new_tokens[t->n] = calloc(nb + 1, sizeof(void*)); // +1: to add also the end
  if (new_tokens[t->n] == NULL) {
    t->tokens = new_tokens; // bonus.
    // And doN'T free t->tokens/new_tokens so as NOT to loose old stuff!
    return 1;
  }

  for (size_t i = 0; i < nb; ++i) {
    new_tokens[t->n][i] = t->s + pos[i];
  }
  // add the end
  new_tokens[t->n][nb] = t->s + strlen(t->s);

  t->tokens = new_tokens;
  t->n = new_n;
  return 0;
}
```

Il faut faire attention de ne pas perdre le contenu original en cas d'échec du `realloc()` (ou des `calloc` des nouveaux tokens).

On a ici la stratégie simple d'augmentation linéaire de la taille (+1 à chaque fois). On pourrait bien sûr sophistiquer avec allocation de plages exponentiellement plus grandes à chaque fois (recherche logarithmique au lieu de linéaire de la taille optimale) ; mais il faudrait alors une variable (statique) ou un champ supplémentaire.

Commentaires de correction : réussie à 65%. Les principales erreurs ont été :

- allocation et affectation du dernier pointeur du tableau manquants.
- Mauvais usage du `realloc` (`ptr = realloc(ptr)`).
- Oublier de créer un espace supplémentaire quand on veut ajouter une tokenization.
- Ne pas vérifier les retour de `*alloc`
- Oublier de multiplier la taille désirée par `sizeof(x)`

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()` :

```c
int add_whitespace_tokenization(tokenizations* t)
{
  if (t == NULL) return 1; // bonus (1 seule fois)

  size_t nb = strlen(t->s); // at most everything is a whitespace
  size_t* const pos = calloc(nb, sizeof(size_t));
  if (pos == NULL) return 1;

  nb = 0; // actual nb of tokens
  for (const char* p = t->s; *p; ++p) {
    if (isspace(*p)) // or *p == ' '
    {  pos[nb++] = (size_t) (p - t->s); // no type problem here: p >= t->s for sure
    }
  }

  const int err = add_tokenization(t, nb, pos);
  free(pos); // don't forget to release it ;-)
  return err;
}
```

On peut bien sûr adopter d'autres stratégies pour l'allocation initiale (p.ex. compter le nombre de blancs).
On peut ici (examen + supposition faible nombre de tokens) tolérer une VLA même si celles-ci sont à proscrire en général.
Et il n'est pas strictement nécessaire d'utiliser ici l'arithmétique des pointeurs, mais c'est tellement plus élégant...

Commentaires de correction : réussie à 70%. Les principales erreurs ont été :

- beaucoup d'allocations inutiles : `malloc` de taille 1 puis un realloc pour chaque nouvel espace trouvé.
- utilisation de VLA (à proscrire)
- oublier de 'free' l'allocation dynamique locale
- copier-coller de `add_tokenization`

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()` :

```c
void release(tokenizations* t)
{
  if (t == NULL) return; // bonus (1 seule fois)

  for (size_t i = 0; i < t->n; ++i) {
    free(t->tokens[i]);
  }
  free(t->tokens);
  t->tokens = NULL;
  t->n = 0;
}
```

Commentaires de correction : réussie à 58%. Les principales erreurs ont été :

- `free` des caractères de la chaîne (un niveau en trop).
- `free` de la chaîne de référence

— oubli de `free` (!) : uniquement affectation à `NULL`
— oubli de reset la structure.