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 final

22 mai 2023

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 (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 ; 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(e) des assistant(e)s.
   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. L'examen comporte deux 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 96 points) ; tous les exercices comptent pour la note finale :
   — question 1 : 63.5 points ;
   — question 2 : 32.5 points ;

Question 1 – Files de priorité [sur 63.5 points]

Le but de cet exercice est d'écrire des parties d'un programme permettant de gérer des files de priorité (*priority queue*) contenant des chaînes de caractères.

Une file de priorité est simplement une liste doublement chaînée [1] où les chaînons/éléments ont une valeur (chaîne de caractères ici) et une priorité (`int` ; on peut avoir des priorités négatives).

1.1 – Types de données [sur 6 points]

Les deux types principaux d'un tel programme sont les chaînons/éléments, que nous appellerons `Node`, et la file de priorité elle-même, que nous appellerons `PQueue`.

Un `Node` contient :

- une valeur (chaîne de caractères), qui lui est propre (copie d'une autre) ;
- une priorité (`int`) ;
- le moyen d'accéder au `Node` précédent et au `Node` suivant dans la file.

Une `PQueue` est très simplement juste le moyen d'accéder au premier `Node`, que l'on appellera `head`, et au dernier `Node`, que l'on appellera `tail`.

Définissez ici le type `Node` :

```
typedef struct Node_ {
    char* label;  // non-const
    int priority;
    struct Node_* prev;
    struct Node_* next;
} Node;
```

Attention à la définition cyclique !

Définissez ici le type `PQueue` :

```
typedef struct {
    Node* head;
    Node* tail;
} PQueue;
```

À la limite la `PQueue` peut contenir deux `Node` (c'est évidemment impossible pour `Node` lui-même !), mais ça rend la gestion de tout le reste bien plus compliquée.

1.2 – Créations et destructions [sur 15 points]

Il faut bien sûr pouvoir créer et supprimer des `Node` et des `PQueue`. On vous demande pour cela de définir quatre fonctions :

[...]

Définissez ici les quatre fonctions demandées :

1. chaque chaînon/élément peut accéder à son prédécesseur et à son successeur

```
Node* create_new_node(const char* label, int priority)
{
    Node* new_node = calloc(1, sizeof(Node));
    if (new_node != NULL) {
        new_node->priority = priority;
        new_node->next = NULL;
        new_node->prev = NULL;

        if (label) {
          new_node->label = calloc(strlen(label) + 1, 1);
          if (new_node->label != NULL) strcpy(new_node->label, label);
        } else { // whatever, even do nothing for the beginning on
          new_node->label = NULL;
        }
    }
    return new_node;
}

PQueue* create_priority_queue(void)
{
    return calloc(1, sizeof(PQueue));
}

void destroy_node(Node* node)
{
    if (node != NULL) {
        free(node->label);
        free(node);
    }
}

void destroy_queue(PQueue* queue)
{
    if (queue == NULL) return;
    Node* ptr = queue->head;
    while (ptr != NULL) {
        Node* this_ptr = ptr;
        ptr = ptr->next;

        destroy_node(this_ptr);
    }
    free(queue);
}
```

1.3 – Insertion d'une valeur [sur 18 points]

On souhaite maintenant pouvoir ajouter des valeurs à la file, avec leur priorité. On vous demande pour cela de définir une fonction

`int insert_value(PQueue* queue, const char* label, int priority)`

qui insère (nouveau `Node`) la valeur `label` **après** le dernier `Node` de priorité supérieure ou égale à `priority`. On supposera `queue` proprement ordonnée [2].

Par exemple, si l'on insère la valeur `"Koala"` de priorité 6 dans la file contenant :

```
(head)  [9] Kangaroo
        [8] Elephant
        [6] Cormorant
(tail)  [5] Penguin
```

alors la file contiendra :

```
(head)  [9] Kangaroo
        [8] Elephant
        [6] Cormorant
        [6] Koala
(tail)  [5] Penguin
```

Bien sûr, l'insertion d'une valeur de priorité strictement plus grande que toutes les priorités déjà présentes se fera en tête (`head`) de file, et l'insertion d'une valeur de priorité plus petite ou égale à toutes les priorités déjà présentes se fera en queue (`tail`) de file.

La recherche de la position (priorité) où ajouter la nouvelle valeur se fera simplement de façon linéaire depuis `head`.

La fonction `insert_value()` retourne 0 si elle n'a pas pu insérer et 1 si c'est tout bon.

Définissez ici la fonction `insert_value()` :

2. Vous n'avez pas à vérifier cette propriété.

```
int insert_value(PQueue* queue, const char* label, int priority)
{
    Node* new_node = create_new_node(label, priority);
    if (new_node == NULL) {
        return 0;
    }

    // Linear search of proper place
    Node* ptr = queue->head;
    while (ptr != NULL && ptr->priority >= priority) {
        ptr = ptr->next;
    }

    // 3 cases: head, tail and middle

    // I need to become the new head
    if (ptr == queue->head) {
        new_node->next = ptr;

        if (queue->head != NULL) {
            queue->head->prev = new_node;
        } else {
            queue->tail = new_node;
        }

        queue->head = new_node;
    } else {
        // In the middle of two nodes
        if (ptr != NULL) {
            new_node->prev = ptr->prev;
            ptr->prev->next = new_node;
            ptr->prev = new_node;
            new_node->next = ptr;
        } else {
            // Reached the end of the queue, add the last element
            new_node->prev = queue->tail;
            if (queue->tail != NULL) {
                queue->tail->next = new_node;
            }
            queue->tail = new_node;
        }
    }

    return 1;
}
```

1.4 – Fusion des chaînes de même priorité [sur 10.5 points]

On souhaite aussi offrir une fonctionnalité qui créée une nouvelle chaîne de caractères (valeur de retour, totalement hors de la `PQueue`) qui est la concaténation [3] de toutes les valeurs (chaînes de caractères) de même priorité.

Par exemple, si la file contient :

```
(head) [18] Unicorn
       [10] Elephant
       [10] Bear
       [10] Duck
       [ 7] Penguin
(tail) [ 5] Seagull
```

et que l'on demande la concaténation des chaînes de priorité 10, la fonction `merge_priority()` retournera la nouvelle chaîne `"ElephantBearDuck"` (dans cet ordre), *sans* modifier la file de priorité. Si l'on demande celle des chaînes de priorité 15, on obtiendra la chaîne vide et si l'on demande celle des chaînes de priorité 5, on obtiendra `"Seagull"`.

Définissez ici la fonction `merge_priority()` correspondant à la description ci-dessus (à vous de proposer son prototype) :

```
char* merge_priority(const PQueue* queue, int priority)
{
  char* result = calloc(1,1);
  size_t len = 0;
  for (Node* ptr = queue->head; ptr != NULL; ptr = ptr->next) {
    if (ptr->priority == priority) {
      // better: make a tool function
      len += strlen(ptr->label);
      char* new = realloc(result, len + 1);
      if (new == NULL) return result;
      result = new;
      strcat(result, ptr->label);
    }
  }
    return result;
}
```

3. c.-à-d. la mise bout à bout

1.5 – Échanges [sur 14 points]

Techniquement, il peut être nécessaire de pouvoir échanger un `Node` avec son prédécesseur dans la file (*sans* se préoccuper de leur priorité, juste les échanger).

Définissez ici la fonction `void swap_with_previous(PQueue* queue, Node* node)` qui pour un `Node` donné dans une `PQueue` donnée, effectue cet échange (avec son prédécesseur, lorsqu'il existe) ; on ne vous demande pas de vérifier que le `Node` est effectivement bien présent dans la `PQueue` :

```
void swap_with_previous(PQueue* queue, Node* node)
{
    if (node == NULL || node->prev == NULL) {
        return;
    }

    Node* previous = node->prev;
    Node* next = node->next;

    // Update head when required
    if (previous == queue->head) {
        queue->head = node;
    }

    // Update tail when required
    if (node == queue->tail) {
        queue->tail = previous;
    }

    node->prev = previous->prev;
    node->next = previous;
    previous->prev = node;
    previous->next = next;

    if (node->prev != NULL) {
        node->prev->next = node;
    }
    if (next != NULL) {
        next->prev = previous;
    }
}
```

Note : il n'est bien sûr pas question ici de faire une recherche linéaire du `Node` donné, ce qui ôte tout l'intérêt de la structure de données.

Question 2 – Petites questions [32.5 points]

2.1 H ? Tag ? ? [3 points]

À quoi sert un fichier `.h` ?

Réponse : Cela sert à fournir ce qui est nécessaire (types, prototypes) aux **autres** `.c` pour compiler.

2.2 Qu'est-ce qu'il me dit ? [4 points]

Lors de la création d'un projet de programmation, vous obtenez l'erreur suivante :

```
/usr/bin/ld : polonaise.o : dans la fonction « eval » :
polonaise.c:33 : référence indéfinie vers « construct_vector »
/usr/bin/ld : polonaise.c:39 : référence indéfinie vers « vector_push »
collect2: error: ld returned 1 exit status
```

① Est-ce une erreur de compilation ou d'édition de liens ?

② Que signifie-t-elle (clairement, en français) ?

③ Comment la corriger ? (quel fichier modifier et comment ?)

① C'est une erreur d'édition de liens...

② ...qui dit que certains morceaux (`construct_vector` et `vector_push`) sont manquants ;
il manque donc visiblement un fichier `.o` dans cette édition de liens (typiquement `vector.o`, vus les noms) ;

③ Il faut corriger la commande d'édition de liens, très certainement dans le `Makefile`, ajouter `vector.o` à la liste des dépendances de la cible (exécutable) qui était en train d'être créer.
Cette erreur n'a strictement *rien* à voir avec des `#include`, lesquels sont utilisés à la *compilation* (cf question précédente) !

2.3 Ça fait quoi (1) ? [5 points]

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   typedef struct {
5       double* array[]; // dynamically allocated array of pointers to double
6       size_t nb;       // number of pointers in the array
7   } Container;
8
9   int main(void)
10  {
11      double x = 98.7;
12      Container c;
13      c.nb = 3;
14      c.array = calloc(c.nb, sizeof(double*));
15      c.array[2] = &x;
16      for (size_t i = 0; i < c.nb; ++i)
17          if (c.array[i] != NULL)
18              printf("%zu: %g\n", i, *c.array[i]);
19      return 0;
20  }
```

① Le code ci-dessus compile-t-il ? Si oui, s'exécute-t-il correctement ?

② Si vous répondez « non » à l'une des questions de ①, proposez, directement sur le code (ou à sa droite), un **minimum** de corrections pour qu'il puisse compiler et s'exécuter correctement.

③ Qu'affiche alors ce code ? Justifiez *brièvement* pourquoi.

Réponses :

① Non ça ne compile pas : `double* array[];` est incorrect en C ; au mieux ce serait un « flexible array member », mais devrait alors être en dernier (et le code du `main()` devrait alors aussi être changé !) ; si on avait voulu une allocation statique, alors il aurait *fallu* donner une taille.

② Il faut simplement remplacer `double* array[];` par `double** array;`
il compile alors et s'exécute sans erreur (le manque de `free()` à la fin n'est pas directement une erreur d'exécution, mais une fuite de mémoire).

③ `2: 98.7`
En effet, `calloc()` alloue la mémoire à zéro, donc tous les pointeurs, sauf celui affecté, sont `NULL`.

suite au dos ☞

2.4 Ça fait quoi (2) ? [14 points]

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   void f(char*** a, char* b, size_t* c) {
5       char** d = realloc(*a, (*c + 1) * sizeof(void*));
6       if (!d) return;
7       *a = d;
8       d[*c] = b;
9       ++*c;
10  }
11
12  char** g(char* a, size_t* b) {
13      char** ret = NULL; *b = 0;
14      for (char* c = a; *c; c++) {
15          if (*c == 'y') {
16              *c++ = '\0';
17               f(&ret, a, b);
18               a = c;
19          }
20      } // voir (4) : faire dessin à ce stade
21      f(&ret, a, b);
22      return ret;
23  }
24
25  void h(char** a) { printf("%s ", *a); }
26
27  int main(void)
28  {
29      char str[] = "quidyeratydemonstrandum";
30      size_t s = 0;
31      char** what = g(str, &s);
32      for (size_t i = 0; i < s; ++i) h(what + i);
33      putchar('\n');
34      free(what);
35      return 0;
36  }
```

① Le code ci-dessus compile-t-il ? Si oui, s'exécute-t-il correctement ?

② Si vous répondez « non » à l'une des questions de ①, proposez, directement sur le code (ou à sa droite), un **minimum** de corrections pour qu'il puisse compiler et s'exécuter correctement.

③ Qu'affiche alors ce code ? Expliquer en français pourquoi (mais voir aussi point ④ suivant).

④ **Dessinez** une image illustrative de la mémoire correspondant à la situation atteinte à la ligne 20 du code ci-dessus. Votre dessin doit comprendre toutes les variables de la fonction `g()` et toutes les variables de la fonction `main()`, sauf `i`.

Réponses :

① oui et oui

② —

③ `quid erat demonstrandum` (avec une espace à la fin)

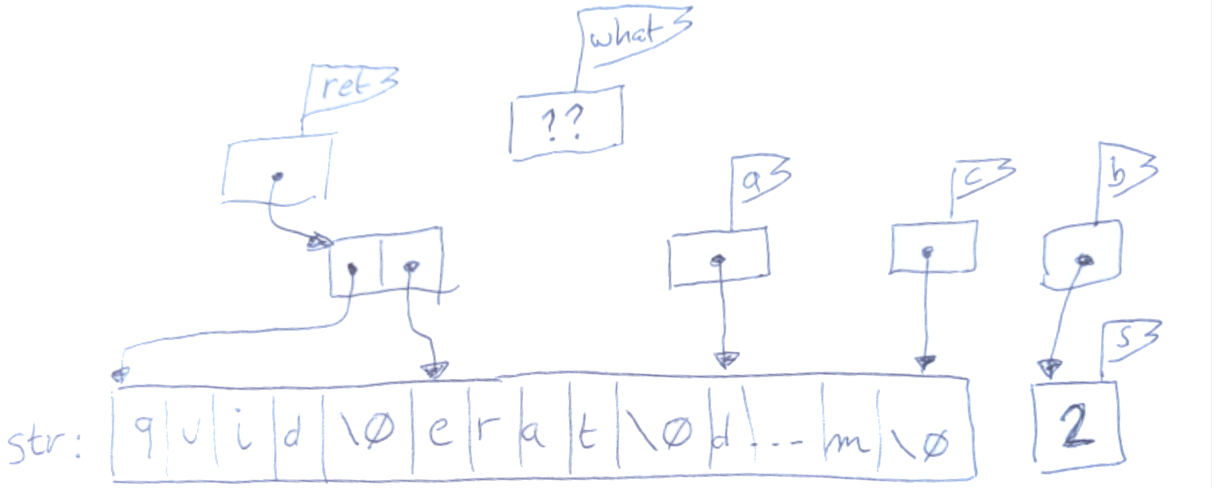
En effet :

— `f()` ajoute une nouvelle chaîne de caractères `b` au tableau de chaines de caractères pointé par `a`, et met à jour (incrémente) `c`

— `g()` tokenize sur-place la chaîne `a` : elle remplace tous les `'y'` par le caractère nul, et stocke les différents tokens dans le tableau (de chaînes de caractères) `ret`

— `h()` affiche simplement la chaîne de caractères pointée par `a`.

④



suite au dos ☞

2.5 Ça fait quoi (3) ? [6.5 points]

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   char* first_two(const char* input)
5   {
6     char str[3] = "";
7     if (input != NULL) {
8       str[0] = input[0];
9       if (input[0] != '\0') str[1] = input[1];
10    }
11    return str;
12  }
13
14  void g(int i)
15  {
16    int a = 2*i;
17    printf("%d\n", a);
18  }
19
20  int main(void)
21  {
22    char* s = first_two("Relax");
23    s[1] = 'a';
24    g(3);
25    printf("%s\n", s);
26    return 0;
27  }
```

① Le code ci-dessus compile-t-il ? Si oui, s'exécute-t-il correctement ?

② Si vous répondez « non » à l'une des questions de ①, proposez, directement sur le code (ou à sa droite), un **minimum** de corrections pour qu'il puisse compiler et s'exécuter correctement.

③ Qu'affiche alors ce code ? Justifiez *brièvement* pourquoi.

Réponses :

① Ça compile mais ne s'exécute pas correctement (pas le résultat désiré).

② Il ne faut jamais retourner d'adresse de variable locale ; il faut allouer dynamiquement `str` dans `first_two()`.
Concrètement :
— il suffit de changer la ligne 6 par : `char* str = calloc(3,1);` ;
— ajouter un `free(s);` en ligne 25.5.

③ Il affichera alors : `6` puis (à la ligne) `Ra` ; la ligne 23 ayant remplacé le `'e'` de `"Relax"` par `'a'` et la fonction `first_two()` ayant mis un caractère nul en troisième position.