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

12 août 2020

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 deux heures pour faire cet examen (16h15 – 18h15).
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 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 100 points) ; tous les exercices comptent pour la note finale :
   — question 1 : 22 points ;
   — question 2 : 54 points ;
   — question 3 : 24 points.

Question 1 Questions brèves [22 points]

Question 1.1 Acquis C ? [8 points]

Expliquez ce qu'affiche le code suivant :

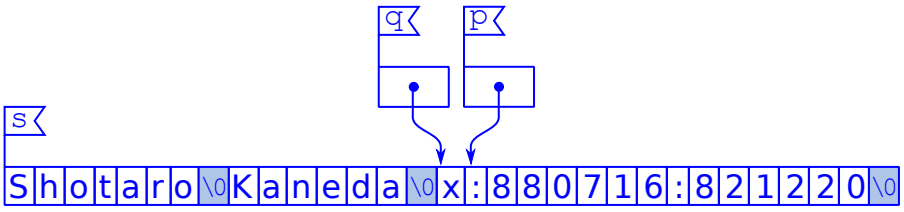
```
char s[] = "Shotaro:Kaneda:x:880716:821220";
for (char *p = s, *q = s; *p; ++p) {
    if (*p == ':') {
        *p = '\0';
        printf("\"%s\", ", q);
        q = p + 1;
    }
}
```

Donnez en particulier un schéma de la mémoire à un moment pertinent en *milieu* d'exécution de la boucle (c.-à-d. ni au début, ni à la fin). Indiquez sur ce schéma le contenu de `s`, ainsi que les pointeurs `p` et `q` à ce moment là.

Réponse : Ce code affiche :
`"Shotaro", "Kaneda", "x", "880716",`

Il remplace en effet itérativement tous les deux-points (`':'`) par des fin de chaîne (`'\0'`), affichant à chaque fois le token en cours (`q`).
A noter, cependant, que le dernier token (`"821220"`) **n'**est **pas** affiché. Il manque son affichage *après* la boucle `for`.

Par exemple :



Commentaires de correction : les principales erreurs ont été :
- afficher le dernier token ;
- ou alors s'arrêter au premier `'0'` (confusion entre `'0'` et `'\0'`).

Question 1.2 Qu'est-ce que C ? [3 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(void* p) {
  if (p == NULL) puts("Cessez !");
  else           puts("C'est C !");
}
int main(void)
{
  int* p = NULL;
  f(&p);
  return 0;
}
```

Réponse : Oui : `C'est C !` ; l'*adresse* de `p` du `main()` n'étant pas `NULL` (et un `int**` peut sans souci être passé comme « pointeur générique» `void*`).

Commentaires de correction : les principales erreurs ont été :
- confusion entre le pointeur et le contenu pointé (entre `p` lui-même (variable) et son contenu (`NULL`)).

Question 1.3 Ah, c'est du 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.
On supposera que `s` est stocké à l'adresse 8899, `"Angus"` à l'adresse 2211 et que le caractère `'s'` se représente par le nombre 115 en mémoire.

```
#include <stdio.h>
int main(void)
{
  const char* s = "Angus";
  printf("%d\n", (int) s[5]);
  return 0;
}
```

Réponse :
Oui : `0`, la valeur entière du *sixième* caractère de `s`, c.-à-d. le `'\0'` final.

Commentaires de correction : les principales erreurs ont été :

- ne pas savoir que `'\0'` est exactement la même chose que `(char) 0`, c.-à-d. un octet dont les 8 bits sont à 0,
- ne pas savoir que `"Angus"` contient *déjà* lui-même le `'\0'` ;
- imaginer que le casting se fait au travers d'un pointeur (c.-à-d. sur la zone mémoire) et non pas sur la valeur, c.-à-d. confondre `(int) c` et `*((int*) pc)`.

Question 1.4 Dans C maintenant. [3 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>
int main(void)
{
  char s[6];
  s = "cigale";
  printf("%s\n", s);
  return 0;
}
```

Réponse :
Non : `s` est un tableau, et on ne peut pas affecter de tableau.

Commentaires de correction : les principales erreurs ont été :

- ne pas savoir qu'on ne peut pas affecter de tableau ;
- ne pas voir que c'est une affectation et non pas une initialisation ;
- penser qu'il y a un problème de taille et surtout que cela entrainerait une erreur de compilation (!) ou d'exécution : si c'était une initialisation au lieu d'une affectation, alors le code compilerait sans soucis (il n'y a pas de vérification de taille à la compilation) et s'exécuterait correctement (avec un buffer overflow de 1) car le `'\0'` serait quand même écrit en mémoire, *sans* segmentation fault.

Question 1.5 C'est à 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>
int main(void)
{
  const char* s = "orque";
  const char* t = s;
  for (size_t i = 0; *t != 0; ++i) {
    t = s + i;
    printf("%s", t);
  }
  return 0;
}
```

Réponse : Oui : `orquerquequeuee`, car chaque position successive dans `s` est affichée comme une chaîne à part entière (à cause du `%s`).

Commentaires de correction : les principales erreurs ont été :

- mal comprendre le `const` : il n'empêche pas l'accès en *lecture* ni n'empêche le changement de la valeur de `t` lui-même (c.-à-d. on peut faire pointer `t` ailleurs ; mais il empêche la modification de la valeur *pointée*) ;
- penser que `*t != 0` est incorrect et devrait être écrit `*t != '\0'` (notez que sur toutes les machines sur lesquelles un « mot mémoire » est strictement plus grand qu'un octet, cette comparaison se fait *de toutes façons* en `int` et non pas en `char`).

Question 2 Gestion d'un disque dur [54 points]

On s'intéresse à la conception (partielle) d'un système de gestion de fichiers sur disque dur.

Un disque dur est décomposé en T secteurs, dont la taille est typiquement de 512 octets (voir fig. 1 ci-contre). Une lecture ou écriture sur un disque ne peut opérer que sur des secteurs entiers. Ainsi, pour écrire le 21212^{e} octet du disque, il faut d'abord lire le secteur 42 (car $21212/512 = 42$), modifier (en mémoire) l'octet position 220 du secteur lu (car $21212 \bmod 512 = 220$), puis réécrire ensuite le secteur ainsi modifié à sa place d'origine (42).

Le disque stocke non seulement le contenu des fichiers eux-mêmes, mais également toutes les « méta-données » permettant de gérer le système de fichiers (cf. fig. 1) : informations sur les fichiers, répertoires, mais aussi les secteurs libres. Pour information (on ne l'utilisera pas), le premier secteur du disque (appelé « superblock ») contient une structure qui mémorise l'emplacement et la taille sur le disque de toutes les méta-données.

Le but de cet exercice est de fournir certaines parties d'un programme de gestion de disque dur dont on possède déjà certaines autres parties. En particulier, le type `hdd_t` est déjà fourni et permet de représenter un disque dur. Ce type possède (entre autres) un champ `sector_size` de type `unsigned short int` contenant la taille d'un secteur en octets (typiquement 512). Ce type possède également un champ `root` de type `size_t` contenant le numéro du secteur utilisé pour stocker le premier fichier (A sur la fig. 1) et un champ `nb_sectors` de type `size_t` contenant le nombres total de secteurs (T sur la fig. 1).

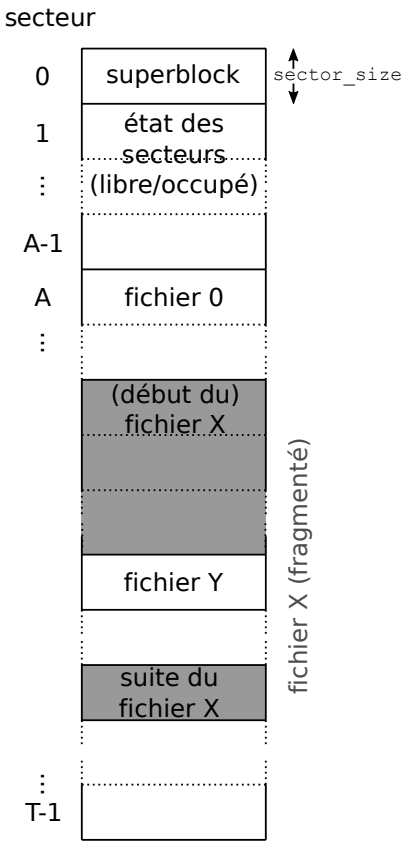


FIGURE 1 – Organisation physique d'un disque dur.

Question 2.1 Lecture de fichiers [14 points]

On suppose également avoir une fonction

```
int hdd_read_sector(char* destination, hdd_t* hdd, size_t sector_number);
```

qui permet de lire en mémoire un secteur donné : elle recopie en mémoire à l'adresse `destination` le contenu du secteur numéro `sector_number` (du disque `hdd`) et retourne 0 si tout s'est bien passé ou un code d'erreur non nul en cas d'erreur.
Note : `destination` est de type `char*` non pas parce que c'est une chaîne de caractères, mais simplement pour lire la mémoire octet par octet (en C, `char` est synonyme d'« octet » (« *byte* », plus exactement) : `sizeof(char) == 1`).

Définissez sur la page ci-contre une fonction `hdd_read_bytes()` qui lit dans une zone mémoire donnée, un nombre spécifié d'*octets* à partir d'un numéro de secteur donné (c'est à vous de décider son prototype). On supposera que la zone mémoire donnée contient la place nécessaire (en clair, ce n'est pas à cette fonction de faire l'allocation).

Indications : le nombre d'octets à lire peut être plus grand que plusieurs secteurs et faites attention au cas où il n'est pas multiple de la taille d'un secteur : la zone mémoire de destination n'a pas la place pour

lire la fin du dernier secteur.

Réponse : Code solution au dos ☞

Plusieurs réponses possibles pour la gestion du débordement :

1. arrondir vers le *haut* le nombre d'octets à lire, allouer un bloc temporaire de cette taille totale (multiple de la taille des secteurs), y lire tous les secteurs, puis copier le nombre original d'octets vers le vrai bloc destination ;
2. ou alors lire les $n-1$ premier secteurs directement dans la mémoire cible ; allouer de la place temporaire pour lire le dernier secteur, et copier `require_sized % sector_size` dans `buffer + required_size / sector_size`.

Je préfère largement la seconde qui nécessite beaucoup moins de mémoire supplémentaire, mais la première est correcte.

Concernant le type de retour, celui est libre mais il faut bien sûr pourvoir être capable de reporter les erreurs d'une façon ou d'une autre.

Note : on n'a pas vu `memcpy()` en cours, mais si certains l'utilisent, pas de problème (au contraire !).

Commentaires de correction : les principales erreurs ont été :

- erreurs de gestion mémoire pour le débordement :
  - non contrôle de la valeur de retour de `malloc` ;
  - oubli de libérer cette mémoire ;
- utilisation de « constantes en dur » (« *magic number* », 512) au lieu de variables ;
- abus d'`assert` pour le contrôle des arguments (il n'est pas bon d'utiliser `assert` pour contrôler des cas d'erreur possibles de fonctions d'interface ; cela peut être toléré pour des fonctions-outils strictement internes mais pas pour des fonctions d'interface) ;
- ne pas écrire à la bonne place : oubli d'incrémenter le pointeur destination ;
- utiliser `strcpy` : cela ne fonctionne pas car s'arrête au premier 0 (zéro) rencontré dans la zone à copier !

```
size_t hdd_read_bytes(char* dest, hdd_t* hdd, size_t start_sector,
                      size_t byte_count)
{
  size_t bytes_read = 0;
  if (hdd  == NULL) return bytes_read;
  if (dest == NULL) return bytes_read;

  size_t sector_count = byte_count / hdd->sector_size;

  while (sector_count != 0) {
    const int error = hdd_read_sector(dest, hdd, start_sector);
    if (error) return bytes_read;

    dest       += hdd->sector_size;
    bytes_read += hdd->sector_size;
    ++start_sector;
    --sector_count;
  }

  const size_t extra_bytes  = byte_count % hdd->sector_size;
  if (extra_bytes != 0) {
    char* extra_read = calloc(hdd->sector_size, 1); // could also be a VLA
    if (extra_read == NULL) return bytes_read;

    const int error = hdd_read_sector(extra_read, hdd, start_sector);
    if (error) {
      free(extra_read); // garbage collecting
      return bytes_read;
    }

    /* shorter:
     *     memcpy(dest, extra_read, extra_bytes);
     * but memcpy may not be known.                    */
    for (size_t i = 0; i < extra_bytes; ++i) {
      dest[i] = extra_read[i];
    }
  }

  free(extra_read); // garbage collecting
  return bytes_read;
}
```

Question 2.2 Recherche de place [25 points]

Les secteurs 1 et suivants (jusque $A-1$ sur la fig. 1) contiennent, bit par bit (dans l'ordre de numéro de bit croissant ; voir la fonction `get_bit()` ci-dessous), le statut libre (0) ou occupé (1) de chacun des secteurs. Ainsi, si le secteur 42 est occupé, le bit 2 ($42 \bmod 8 = 2$) de l'octet 5 ($42/8 = 5$) du secteur 1 ($42/4096 = 0$) [1] sera à 1 ; si le secteur 123456789 (disque d'au moins 60 Go) est libre, le bit 5 ($123456789 \bmod 8 = 5$) de l'octet 418 ($(123456789 \bmod 4096)/8 = 418$) du secteur 30141 ($123456789/4096 = 30140$) sera à 0. [2]

On suppose avoir à disposition une fonction

```
int get_bit(char* p, unsigned int index);
```

qui retourne la valeur (0 ou 1) du bit d'index « `index` » dans l'octet pointé par `p`.

Définissez une fonction `hdd_get_free_sectors()` qui retourne l'emplacement du premier secteur libre, ainsi que le nombre de secteurs libres contigus (qui le suivent, lui compris). Expliquez les paramètres ainsi que l'éventuelle valeur de retour.

Indication : pensez à modulariser : déléguez à une sous-tâche (qu'il vous faudra aussi écrire) déterminant si un secteur est libre ou non. On ne cherchera pas ici à minimiser le nombre d'accès au disque [3].

Réponse : Voici le squelette principal de ce qui est attendu (version plus simple, mais incomplète car ne gérant pas les cas d'erreur interne) :

```
/* Retuns 0 (sector is not free),
 *        1 (sector is free).     */
int is_free(hdd_t* hdd, size_t sector)
{
  char sector_read[hdd->sector_size];
  const size_t metadata_sector_size = 8 * hdd->sector_size;
  hdd_read_sector(sector_read, hdd, sector / metadata_sector_size + 1);
  return 1 - get_bit(sector_read + ((sector % metadata_sector_size) / 8), sector % 8);
}

// ---------------------------------------------------------------------
int hdd_get_free_sectors(hdd_t* hdd, size_t* first_sector, size_t* sector_count)
{
  // search for the first free sector
  *first_sector = hdd->root;
  while ((*first_sector < hdd->nb_sectors) && !is_free(hdd, *first_sector)) (*first_sector)++;
  if (*first_sector >= hdd->nb_sectors) return 1; // disk is full (whatever error code)

  // search for the next non free sector
  size_t sector_2 = *first_sector + 1;
  while ((sector_2 < hdd->nb_sectors) && is_free(hdd, sector_2)) sector_2++;

  *sector_count = sector_2 - *first_sector;

  return 0;
}
```

1. $4096 = 8 \times 512$.
2. En réalité, on peut faire un peu mieux et ne pas coder le fait que les A premiers secteurs sont de toutes façons occupés ; mais ceci serait un peu trop compliqué pour cet examen.
3. Ceci pourrait être fait par la suite (cache) et n'est pas du tout considéré dans cet examen.

La version correcte avec gestion des erreurs serait alors :

```
/* Retuns 0 (sector is not free),
 *        1 (sector is free)
 *     or 2 (sector status is undefined due to errors). */
int is_free(hdd_t* hdd, size_t sector)
{
  // memory place to read one sector
  char* sector_read = malloc(hdd->sector_size);
  if (sector_read == NULL) return 2; // memory error

  const size_t metadata_sector_size = 8 * hdd->sector_size;
  const size_t metadata_sector = sector / metadata_sector_size + 1;
  int status = hdd_read_sector(sector_read, hdd, metadata_sector);
  if (status) {
    status = 2;
  } else {
    const size_t metadata_byte = (sector % metadata_sector_size) / 8;
    status = 1 - get_bit(sector_read + metadata_byte, sector % 8);
  }

  // garbage collecting
  free(sector_read);

  return status;
}

// ----------------------------------------------------------------------
int hdd_get_free_sectors(hdd_t* hdd, size_t* first_sector, size_t* sector_count)
{
  if ((hdd == NULL)
      || (first_sector == NULL)
      || (sector_count == NULL)) return 1; // bad arg (whatever error code)

  // search for the first free sector
  size_t sector_1 = hdd->root;
  int status = 0;
  while ((sector_1 < hdd->nb_sectors) && ((status = is_free(hdd, sector_1)) == 0)) {
    sector_1++;
  }
  if (status == 2) return 2; // IO error (whatever error code)
  if (sector_1 >= hdd->nb_sectors) return 4; // disk is full (whatever error code)

  // search for the next non free sector
  size_t sector_2 = sector_1 + 1;
  while ((sector_2 < hdd->nb_sectors) && ((status = is_free(hdd, sector_2)) == 1)) {
    sector_2++;
  }
  if (status == 2) return 2; // IO error (whatever error code)

  // assign output
  *first_sector = sector_1;
  *sector_count = sector_2 - sector_1;

  return 0;
}
```

Bien sûr, plusieurs autres implémentations et prototypes sont possibles.
Par exemple, il n'est pas nécessaire ici d'avoir de type de retour pour indiquer l'erreur puisque le `first_sector` ne peut jamais être 0 (super block) ; on peut donc utiliser cette valeur pour indiquer une erreur.
De même, par définition même, `count` doit aussi être supérieur ou égal à 1 sauf si le disque est plein. Ce qui au final nous laisse trois codes d'erreur possibles.
Mais, quelque soit le prototype choisi, il est nécessaire de pouvoir tester l'échec d'une façon ou d'une autre.
Par ailleurs, `first_sector` et `sector_count` peuvent très bien être regroupés dans une `struct` qui est retournée. Enfin, si ça en aide certain(e)s, on peut utiliser la fonction `hdd_read_bytes()` au lieu de `hdd_read_sector()`.

Commentaires de correction : ceci était **la** question difficile de l'examen. Les principales erreurs ont été :

- ne pas commencer à controler les secteurs libres à partir de la racine (« root ») ;
- ne pas s'arrêter de contrôler si les secteurs sont libres avant la fin des secteurs (boucle infinie) ;
- retour de pointeur de variable locale ! !
- pas de lecture des secteurs de métadonnées ;
- modulariser parce que ça a été demandé et non pas pour se simplifier la tâche, ou alors ne pas modulariser du tout ;
- ne pas compter le nombre de secteurs libres contigus mais s'arrêter au premier ;
- ne pas traiter les cas d'erreur, ignorer les retours des fonctions appelées ;
- ne pas traiter le cas du disque plein ;
- utilisation du « *magic number* » 4096 au lieu de `8 * hdd->sector_size` (pour ceux qui veulent pousser encore plus loin : `8` devrait en fait être `CHAR_BIT`).

Question 2.3 Fichiers fragmentés [15 points]

Il se peut que le disque ne contienne pas suffisamment de place contiguë pour stocker un fichier en un seul morceau sur des secteurs successifs (voir le fichier X sur la fig. 1). Ceci arrive en particulier lorsque les fichiers sont créés et supprimés dans le désordre. L'espace libre devient alors « fragmenté ». Dans ce cas, il est nécessaire de mémoriser le fichier en le divisant en plusieurs morceaux (fragments) ayant chacun un emplacement (numéro de secteur) et une taille (en octets).

Le système de fichiers doit donc pouvoir gérer une liste de fragments pour chaque fichier. Définissez (au moins) une structure `file_piece_t` permettant de le faire. Vous pouvez définir d'autres types intermédiaires si vous le souhaitez.

Définissez ensuite une fonction qui ajoute à une telle liste un nouveau fragment (débutant au prochain secteur libre ; aucune écriture n'est faite sur le disque ici).

Indications : cette fonction peut utiliser la fonction de la question précédente ; et pensez à gérer les cas d'erreur.

Réponse : (voir page suivante)

```c
typedef struct file_piece_ {
  size_t first_sector;      // first sector of the piece
  size_t sector_count;      // piece size
  struct file_piece_* next; // next piece
} file_piece_t;

/* Prend en paramètres un disque et le dernier fragment du fichier.          *
 * Retourne le fragment de queue ajouté (qui permettra ainsi un enchaînement *
 * d'appels à add_fragment()); NULL en cas d'erreur.                         */

file_piece_t* add_fragment(hdd_t* hdd, file_piece_t* file_tail)
{
  if ((hdd == NULL) || (file_tail == NULL)) return NULL;

  // creates new fragment to be inserted
  file_piece_t* new = malloc(sizeof(*new));
  if (new != NULL) {
    const int error = hdd_get_free_sectors(hdd                  ,
                                           &(new->first_sector) ,
                                           &(new->sector_count) );
    if (error) {
      free(new);
      return NULL;
    }

    new->next = NULL;

    // insertion
    file_tail->next = new;
  }

  return new;
}
```

D'autres versions sont bien sûr possibles, en particulier si l'on passe la tête de la chaîne des fragments, il faudra alors, bien sûr, avoir la recherche de sa fin avant l'insertion.

De même, plusieurs types de retour sont possibles, mais il est nécessaire de pouvoir tester l'échec d'une façon ou d'une autre.

Concernant le type `file_piece_t`, on peut tolérer que celui-ci soit la *liste* elle-même (pointeur ; au lieu du « chaînon » comme ici), mais il faut alors, bien sûr, un autre type pour le « chaînon » lui-même.

Une implémentation à base de tableau dynamique est aussi correcte (vu qu'on n'ajoute qu'à la fin).

Commentaires de correction : étrangement peu d'étudiant(e)s ont abordé cette question, pourtant plus facile à notre avis. Parmi celles/ceux qui l'ont fait, une implémentation sous forme de tableau dynamique a été majoritairement préférée à celle sous forme de liste chaînée. Les principales erreurs ont été :

- (tableau dynamique) ne pas garder trace de la taille allouée ;
- (tableau dynamique) perte du pointeur initial en cas d'échec de réallocation.

Question 3 Découpage de chaînes de caractères [24 points]

Le but de cet exercice est d'écrire une fonction `split()` qui retourne tous les constituants (« *tokens* ») séparés par un séparateur donné dans une chaîne de caractères donnée.

Par exemple :

- `split('/', "un/exemple/simple")` retournera un tableau contenant les trois chaînes `"un"`, `"exemple"` et `"simple"` ;
- `split('/', "/autre/exemple")` retournera un tableau contenant les trois chaînes `""` (chaîne vide), `"autre"` et `"exemple"` ;
- `split('/', "/autre/exemple/")` retournera un tableau contenant les *quatre* chaînes `""` (chaîne vide), `"autre"`, `"exemple"` et `""` (chaîne vide) ;
- `split('/', "/autre//exemple")` retournera un tableau contenant les *quatre* chaînes `""` (chaîne vide), `"autre"`, `""` (chaîne vide) et `"exemple"` ;
- enfin, `split('/', "autre exemple")` retournera un tableau contenant la seule chaîne `"autre exemple"`.

Question 3.1 Type de retour [4 points]

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

Réponse :

```
typedef struct {
  size_t size;
  const char** content;
} string_array;
```

Commentaires de correction : les principales erreurs ont été :

- ne pas pouvoir connaître la fin du tableau (une taille étant le plus simple) ;
  certain(e)s semblent d'ailleurs penser que l'on peut connaître la taille d'un tableau avec `sizeof` ;
- utilisation de constantes totalement arbitraires.

Question 3.2 Affichage [6 points]

Définissez ici (et sur la page d'en face) une fonction `affiche()` qui affiche une valeur du type que vous venez de définir (Question 3.1). Par exemple, l'appel à `affiche()` sur le résultat de `split('/', "un/exemple/simple")`, affichera :

```
"un", "exemple", "simple",
```

Vous êtes libre de choisir ses paramètres, mais, comme `printf()`, cette fonction devra retourner le nombre total de caractères affichés.

Réponse : (au dos)

Commentaires de correction : les principales erreurs ont été :

- ne pas comprendre la valeur de retour (et donc ne pas connaître celle de `printf`).

```
int print(const string_array* t)
{
  int count = 0;
  if (t != NULL) {
    for (size_t i = 0; i < t->size; ++i) {
      count += printf("\"%s\", ", t->content[i]);
    }
  }
  return count;
}
```

Question 3.3 `split()` [14 points]

Définissez ici (et au dos si nécessaire) la fonction `split()`.

Réponse :

cf Question 1.1 ;-)

Code solution au dos.

Commentaires de correction : les principales erreurs ont été :

- ne pas passer le paramètre comme `const`, ce qui empêche du coup de pouvoir l'appeler avec des valeurs littérales, tel que : `split(':', "abc:defg")` ;
- modification de la chaîne reçue (parfois même lorsque celle-ci était déclarée comme `const`) ;
- ne pas gérer correctement le dernier token ;
- pas du tout d'allocation de mémoire (!) ;
- ne pas terminer les tokens par `'\0'`,
- confusion entre l'allocation de `string_array` et celle de son `content`.

```
int add(string_array* t, const char* s)
{
  if ((t->size % PAGE_SIZE) == 0) {
    // page allocation is optional, can be done 1 by 1
    const char** const new_content = realloc(t->content, t->size + PAGE_SIZE);
    if (new_content == NULL) return 1;
    t->content = new_content;
  }
  t->content[t->size++] = s;
  return 0;
}

void in_place_split(char sep, char* s, string_array* t)
{
  for (char* p = s; *p; ++p) {
    if (*p == sep) {
      *p = '\0';
      add(t, s);
      s = p + 1;
    }
  }
  add(t, s); // don't forget last token!
}

string_array split(char sep, const char* s)
{
  string_array return_value;
  memset(&return_value, 0, sizeof(return_value)); // tableau vide

  if (s != NULL) {
    char* copy = malloc(strlen(s) + 1);
    if (copy != NULL) {
      strcpy(copy, s);
      in_place_split(sep, copy, &return_value);
    }
  }

  return return_value;
}
```