EPITA S4 promo 2019
Programmation - Épreuve machine

Marwan Burelle *

Instructions :

Vous devez lire l'intégralité du sujet, ainsi que l'ensemble de ces instructions. Tout ce qui est explicitement demandé dans ce sujet est obligatoire. La perte de points dûe au non respect des consignes explicites ne sera pas contestable.

Votre répertoire pendant l'épreuve est temporaire, dans ce répertoire vous trouverez un répertoire `subject` *et un répertoire* `submission`*. Dans le répertoire* `subject` *vous trouverez un sous-répertoire* `Skel` *vous devez copier le* **contenu** *de ce répertoire dans votre répertoire de rendu (*`submission`*.)*

Vous devez effectuer des rendus réguliers pour être sûr de ne pas perdre votre travail. Pour effectuer un rendu, il vous suffit d'appeler la commande `submission`.

À titre de rappel, voici les commandes pour commencer votre épreuve :

```
> cd
> cd subject
> cp -R Skel/* ~/submission/
```

Dans ce répertoire vous trouverez : un fichier `Makefile` *permettant de compiler votre code, un certain nombre de fichiers annexes, un fichier pour chaque question de la forme* `questionXX.c`*. Seuls les fichiers de question peuvent être modifiés, les autres seront remplacés lors de la correction par les originaux.*

Le `Makefile` *permet d'engendrer un petit programme de test pour chaque question. Le programme de test se charge des interactions avec l'utilisateur et appelle votre fonction avec les paramètres attendus. Pour obtenir le programme de test de chaque question il faut faire la commande :* `make questionXX`*.*

La compilation lors de la correction utilisera ce `Makefile`*, par conséquent vous n'aurez les points à la question XX que si la commande "*`make questionXX`*" réussit* ***et bien sûr que le résultat est correct****.*

Le barème est donné à titre indicatif et peut être sujet à modifications.

À la fin de ce document vous trouverez des annexes décrivants quelques consignes sur les programmes de tests.

Il y a 24 points et 17 questions dans cet examen.

*marwan.burelle@lse.epita.fr

Question 1 (1)

Écrire la(les) fonction(s) suivante(s):

```
unsigned long fact(unsigned long n);
```

`fact(n)` calcule factorielle de n.

On rappelle que la suite factorielle est définie par :

$$\texttt{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \texttt{fact}(n-1) & \text{otherwise} \end{cases}$$

Exemple 1.1:

```
shell> ./question01 0 5
./question01 0 5
Fixed tests:
fact( 0) = 1
fact( 1) = 1
fact( 2) = 2
fact( 3) = 6
fact( 4) = 24
fact( 5) = 120
Random tests:
fact( 7) = 5040
fact( 6) = 720
fact( 9) = 362880
fact( 3) = 6
fact( 1) = 1
```

Question 2 (1)

Écrire la(les) fonction(s) suivante(s):

```
unsigned long my_intsqrt(unsigned long n);
```

`my_intsqrt(n)` calcule la racine carrée entière de n.

La racine carrée entière est la solution (entière) x à l'inéquation :

$$x^2 \leq n < (x+1)^2$$

Pour résoudre ce problème on utilise la méthode de Héron (une variante de la méthode de Newton). On considère une première approximation x de la racine (cette approximation doit être supérieur à la racine et inférieur à n, on prendra donc n comme première valeur). On calcule la prochaine approximation comme étant la moyenne arithmétique entre x et n/x et on continue tant que x est plus grand que n/x (c'est à dire tant que x est plus grand que la racine cherchée).

Exemple 2.1:

```
shell> ./question02 0 5
Fixed tests:
        my_intsqrt(    0)       =      0        [OK]
        my_intsqrt(    1)       =      1        [OK]
        my_intsqrt(    4)       =      2        [OK]
        my_intsqrt(   16)       =      4        [OK]
        my_intsqrt(   64)       =      8        [OK]
        my_intsqrt(  256)       =     16        [OK]
        my_intsqrt( 1024)       =     32        [OK]
        my_intsqrt( 4096)       =     64        [OK]
        my_intsqrt(16384)       =    128        [OK]
        my_intsqrt(65536)       =    256        [OK]
Random tests:
        my_intsqrt(1804289383)  =  42476        [OK]
        my_intsqrt( 846930886)  =  29102        [OK]
        my_intsqrt(1681692777)  =  41008        [OK]
        my_intsqrt(1714636915)  =  41408        [OK]
        my_intsqrt(1957747793)  =  44246        [OK]
```

Question 3 (1)

Écrire la(les) fonction(s) suivante(s):

```
size_t array_count_occurences(int *begin, int *end, int x);
```

`array_count_occurences(begin, end, x)` compte le nombre de fois où la valeur `x` apparaît dans le tableau entre `begin` (inclus) et `end` (exclus.)

Exemple 3.1:

```
shell> ./question03 1 10
Fixed tests:
  array:
    |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 |
  array_count_occurences(begin, end, 5): 1
Random tests:
  array:
    |  0 |  9 |  8 |  5 |  1 |  8 |  4 |  7 |  5 |  7 |
  array_count_occurences(begin, end, 8): 2
  array_count_occurences(begin, end, 9): 1
  array_count_occurences(begin, end, 42): 0
```

Question 4 (1)

Écrire la(les) fonction(s) suivante(s):

```
int* array_top(int *begin, int *end);
```

`array_top(begin, end)` vérifie si le tableau entre `begin` (inclus) et `end` (exclus) est une *colline* et renvoie le sommet de celle-ci. Un tableau est une colline si il est divisé en une partie croissante puis une partie décroissante. Un tableau trié intégralement est aussi une colline. Le sommet correspond à l'élément le plus grand (donc celui ou la direction s'inverse.)

Si le tableau est une colline, la fonction renverra le pointeur sur la case correspondant au sommet ou `NULL` si le tableau n'est pas une colline.

Exemple 4.1:

```
shell> ./question04 1 7
Hill array tests:
  array:
    |   0 |   1 |   2 |   3 |   4 |   3 |   2 |
array_top(begin, end) = 4 (OK)
Random tests:
  array:
    | 719 | 788 | 575 |  61 | 498 | 864 | 277 |
array_top(begin, end) returned NULL
```

Question 5 (1)

Écrire la(les) fonction(s) suivante(s):

```
void insert_sort(int *begin, int *end);
```

`insert_sort(begin, end)` trie le tableau entre `begin` (inclus) et `end` (exclus.) On utilisera un tri par insertion, dont voici l'algorithme :

```
insert_sort(tab, left, right):
  for cur = left to right - 1 do:
    x = tab[cur]
    i = cur
    while i > left && x < tab[i - 1] do:
      tab[i] = tab[i - 1]
      i = i - 1
    done
    tab[i] = x
  done
```

Exemple 5.1:

```
shell> ./question05 0 5
./question05 0 5
Fixed tests:
  sorted array:
    |   1 |   2 |   3 |   4 |   5 |
  sorting ... [OK]
  after insert_sort:
    |   1 |   2 |   3 |   4 |   5 |

  reverse sorted array:
    |   5 |   4 |   3 |   2 |   1 |
  sorting ... [OK]
  after insert_sort:
    |   1 |   2 |   3 |   4 |   5 |

Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 |
  sorting ... [OK]
  after insert_sort:
    | 383 | 777 | 793 | 886 | 915 |
```

Question 6 (1)

Écrire la(les) fonction(s) suivante(s):

```
size_t mystrlen(char *s);
```

`mystrlen(s)` renvoie le nombre de caractères de la chaîne `s` (le pointeur ne sera pas `NULL`.) Vous devrez respecter le comportement attendu pour la fonction `strlen(3)`.

Exemple 6.1:

```
shell> ./question06 0 5
s = "n{6\P"
mystrlen(s) = 5 -- check: [OK]
```

Question 7 (1)

Écrire la(les) fonction(s) suivante(s):

```
char *mystrncpy(char *dst, char *src, size_t len);
```

`mystrncpy(dst,src,len)` : copie au plus `len` caractères de la chaîne `src` dans la chaîne `dst`. On suppose que `src` et `dst` sont non `NULL`, `src` est terminée par un `'\0'` et `dst` est de taille suffisante.

Dans tous les cas, `mystrncpy(dst,src,len)` écrit exactement `len` caractères dans `dst`. Si le nombre de caractères de `src` est inférieur à `len`, alors votre fonction devra remplir la fin de `dst` par des `'\0'`. Sinon (`src` plus grand que `len`) votre fonction ne doit pas mettre de `'\0'` à la fin de `dst` (voir `strncpy(3)`.)

Je vous conseille fortement de lire la page de manuel de la fonction `strncpy(3)` qui fournit une description complète de cette fonction.

Exemple 7.1:

```
shell> ./question07 0 10
src = "n{6\Pavw[:"

test: mystrncpy(dst,src,11)
dst = "n{6\Pavw[:"
-- check:
  first char:  [OK]
  last char:  [OK]
  0 fill:  [OK]
  overflow:  [OK]

test: mystrncpy(dst,src,5)
dst = "n{6\P"
-- check:
  first char:  [OK]
  last char:  [OK]
  overflow:  [OK]

test: mystrncpy(dst,src,20)
dst = "n{6\Pavw[:"
-- check:
  first char:  [OK]
  last char:  [OK]
  0 fill:  [OK]
  overflow:  [OK]

test: mystrncpy(dst,src,0)
-- check:
  overflow:  [OK]
```

Question 8 (1)

Écrire la(les) fonction(s) suivante(s):

```c
size_t list_len(struct list *list);
```

`list_len(list)` calcule la taille de la liste `list`. La liste en entrée dispose d'une sentinelle qui ne doit pas être comptée dans la taille de la liste.

Le type représentant les listes est le suivant :

```c
struct list {
  struct list *next;
};
```

Exemple 8.1:

```
shell> ./question08 0 5
Fixed tests:
  list_len(list) = 5  [OK]
```

Question 9 (1)

Écrire la(les) fonction(s) suivante(s):

```c
size_t list_hist(struct list *list, size_t hist[256]);
```

`list_hist(list, hist)` calcule l'histogramme des valeurs de la liste `list`. Un histogramme des valeurs est tableau qui résume le nombre d'occurrences de chaque valeur possible dans la liste. Dans notre cas, les valeurs possible sont au nombre de 256 (un seul octet non signé). La fonction remplira le tableau `hist` déjà alloué à la bonne taille mais pas initialisé. La fonction renvoit également le compte des éléments de la liste.

Le type représentant les listes est le suivant :

```c
struct list {
  struct list *next;
  unsigned char value;
};
```

Exemple 9.1:

```
shell> ./question09 0 20
Random tests:
list = -> 198 -> 105 -> 115 ->  81 -> 255 ->  74 -> 236 ->  41 -> 205 -> 186
       -> 171 -> 242 -> 251 -> 227 ->  70 -> 124 -> 194 ->  84 -> 248 ->   0
|hist[  0]: 1|hist[ 64]:  |hist[128]:  |hist[192]:  |
|hist[  1]:  |hist[ 65]:  |hist[129]:  |hist[193]:  |
|hist[  2]:  |hist[ 66]:  |hist[130]:  |hist[194]: 1|
|hist[  3]:  |hist[ 67]:  |hist[131]:  |hist[195]:  |
```

```
|hist[  4]:  |hist[ 68]:  |hist[132]:  |hist[196]:  |
|hist[  5]:  |hist[ 69]:  |hist[133]:  |hist[197]:  |
|hist[  6]:  |hist[ 70]: 1|hist[134]:  |hist[198]: 1|
|hist[  7]:  |hist[ 71]:  |hist[135]:  |hist[199]:  |
|hist[  8]:  |hist[ 72]:  |hist[136]:  |hist[200]:  |
|hist[  9]:  |hist[ 73]:  |hist[137]:  |hist[201]:  |
|hist[ 10]:  |hist[ 74]: 1|hist[138]:  |hist[202]:  |
|hist[ 11]:  |hist[ 75]:  |hist[139]:  |hist[203]:  |
|hist[ 12]:  |hist[ 76]:  |hist[140]:  |hist[204]:  |
|hist[ 13]:  |hist[ 77]:  |hist[141]:  |hist[205]: 1|
|hist[ 14]:  |hist[ 78]:  |hist[142]:  |hist[206]:  |
|hist[ 15]:  |hist[ 79]:  |hist[143]:  |hist[207]:  |
|hist[ 16]:  |hist[ 80]:  |hist[144]:  |hist[208]:  |
|hist[ 17]:  |hist[ 81]: 1|hist[145]:  |hist[209]:  |
|hist[ 18]:  |hist[ 82]:  |hist[146]:  |hist[210]:  |
|hist[ 19]:  |hist[ 83]:  |hist[147]:  |hist[211]:  |
|hist[ 20]:  |hist[ 84]: 1|hist[148]:  |hist[212]:  |
|hist[ 21]:  |hist[ 85]:  |hist[149]:  |hist[213]:  |
|hist[ 22]:  |hist[ 86]:  |hist[150]:  |hist[214]:  |
|hist[ 23]:  |hist[ 87]:  |hist[151]:  |hist[215]:  |
|hist[ 24]:  |hist[ 88]:  |hist[152]:  |hist[216]:  |
|hist[ 25]:  |hist[ 89]:  |hist[153]:  |hist[217]:  |
|hist[ 26]:  |hist[ 90]:  |hist[154]:  |hist[218]:  |
|hist[ 27]:  |hist[ 91]:  |hist[155]:  |hist[219]:  |
|hist[ 28]:  |hist[ 92]:  |hist[156]:  |hist[220]:  |
|hist[ 29]:  |hist[ 93]:  |hist[157]:  |hist[221]:  |
|hist[ 30]:  |hist[ 94]:  |hist[158]:  |hist[222]:  |
|hist[ 31]:  |hist[ 95]:  |hist[159]:  |hist[223]:  |
|hist[ 32]:  |hist[ 96]:  |hist[160]:  |hist[224]:  |
|hist[ 33]:  |hist[ 97]:  |hist[161]:  |hist[225]:  |
|hist[ 34]:  |hist[ 98]:  |hist[162]:  |hist[226]:  |
|hist[ 35]:  |hist[ 99]:  |hist[163]:  |hist[227]: 1|
|hist[ 36]:  |hist[100]:  |hist[164]:  |hist[228]:  |
|hist[ 37]:  |hist[101]:  |hist[165]:  |hist[229]:  |
|hist[ 38]:  |hist[102]:  |hist[166]:  |hist[230]:  |
|hist[ 39]:  |hist[103]:  |hist[167]:  |hist[231]:  |
|hist[ 40]:  |hist[104]:  |hist[168]:  |hist[232]:  |
|hist[ 41]: 1|hist[105]: 1|hist[169]:  |hist[233]:  |
|hist[ 42]:  |hist[106]:  |hist[170]:  |hist[234]:  |
|hist[ 43]:  |hist[107]:  |hist[171]: 1|hist[235]:  |
|hist[ 44]:  |hist[108]:  |hist[172]:  |hist[236]: 1|
|hist[ 45]:  |hist[109]:  |hist[173]:  |hist[237]:  |
|hist[ 46]:  |hist[110]:  |hist[174]:  |hist[238]:  |
|hist[ 47]:  |hist[111]:  |hist[175]:  |hist[239]:  |
|hist[ 48]:  |hist[112]:  |hist[176]:  |hist[240]:  |
|hist[ 49]:  |hist[113]:  |hist[177]:  |hist[241]:  |
|hist[ 50]:  |hist[114]:  |hist[178]:  |hist[242]: 1|
|hist[ 51]:  |hist[115]: 1|hist[179]:  |hist[243]:  |
|hist[ 52]:  |hist[116]:  |hist[180]:  |hist[244]:  |
|hist[ 53]:  |hist[117]:  |hist[181]:  |hist[245]:  |
|hist[ 54]:  |hist[118]:  |hist[182]:  |hist[246]:  |
|hist[ 55]:  |hist[119]:  |hist[183]:  |hist[247]:  |
|hist[ 56]:  |hist[120]:  |hist[184]:  |hist[248]: 1|
|hist[ 57]:  |hist[121]:  |hist[185]:  |hist[249]:  |
```

```
|hist[ 58]:  |hist[122]:  |hist[186]: 1|hist[250]:  |
|hist[ 59]:  |hist[123]:  |hist[187]:  |hist[251]: 1|
|hist[ 60]:  |hist[124]: 1|hist[188]:  |hist[252]:  |
|hist[ 61]:  |hist[125]:  |hist[189]:  |hist[253]:  |
|hist[ 62]:  |hist[126]:  |hist[190]:  |hist[254]:  |
|hist[ 63]:  |hist[127]:  |hist[191]:  |hist[255]: 1|
summary count: 20
output len: 20
```

Question 10 (2)

Écrire la(les) fonction(s) suivante(s):

```
void list_insert(struct list *list, struct list *elm);
```

`list_insert(list,elm)` insert le maillon (déjà alloué) `elm` dans la liste chaînée `list` à sa place. La liste est triée en ordre croissant et dispose d'une sentinelle.
Le type représentant les listes est le suivant :

```
struct list {
  struct list *next;
  int          val;
};
```

Exemple 10.1:

```
shell> ./question10 0 5
Fixed tests:
list = EMPTY
  insert 1 in list
  insert 2 in list
  insert 3 in list
  insert 4 in list
  insert 5 in list
list = ->   1 ->   2 ->   3 ->   4 ->   5
Random tests:
list = EMPTY
  insert 383 in list
  insert 886 in list
  insert 777 in list
  insert 915 in list
  insert 793 in list
list = -> 383 -> 777 -> 793 -> 886 -> 915
```

Question 11 (1)

Écrire la(les) fonction(s) suivante(s):

```
void list_reverse(struct list *list);
```

`list_reverse(list)` renverse en place la liste `list`. Cette liste a une sentinelle qui doit être laissée en tête de la liste résultat.

Attention : il s'agit d'une liste intrusive (en gros vous n'avez pas accès au contenu) vous devez donc bien détacher les maillons de la liste et les racrocher en ordre inverse.

Le type représentant les listes est le suivant :

```
struct list {
  struct list *next;
};
```

Exemple 11.1:

```
shell> ./question11 0 5
Fixed tests:
list = ->   1 ->   2 ->   3 ->   4 ->   5
list_reverse(list)
list = ->   5 ->   4 ->   3 ->   2 ->   1
Random tests:
list = -> 793 -> 915 -> 777 -> 886 -> 383
list_reverse(list)
list = -> 383 -> 886 -> 777 -> 915 -> 793
```

Question 12 (2)

Écrire la(les) fonction(s) suivante(s):

```
struct matrix *matrix_mul(struct matrix *A, struct matrix *B);
```

`matrix_mul(A, B)` calcule le produit des matrices `A` et `B`. Le résultat est une nouvelle matrice qui devra être correctement créée et allouée.

Les matrices sont représentée par le type suivant :

```
struct matrix {
  size_t lines, cols;
  int   *data;
};
```

De manière évidente, `lines` représente le nombre de lignes de la matrice et `cols` son nombre de colonnes. Le champ `data` est un pointeur sur une zone mémoire contenant `lines * cols` entiers correspondant aux cases de la matrice stockées en mode *ligne d'abord*.

Pour accéder à la case (i, j) de la matrice `A`, on utilisera le décallage classique : `A->data[i * A->cols + j]`.
On rappelle que le produit d'une matrice A de dimensions $n \times m$ par une matrice B de dimensions $m \times p$ est la matrice AB de dimension $(n \times p)$, dont les cases sont données par la formule suivante :

$$AB_{i,j} = \Sigma_{k=0}^{m-1} A_{i,k} \times B_{k,j}$$

Exemple 12.1:

```
shell> ./question12 0 3 5
Test with id matrix:
A =
|  83 |  86 |  77 |
|  15 |  93 |  35 |
|  86 |  92 |  49 |
id =
|   1 |   0 |   0 |
|   0 |   1 |   0 |
|   0 |   0 |   1 |
C = A * id
|  83 |  86 |  77 |
|  15 |  93 |  35 |
|  86 |  92 |  49 |
Random tests:
A =
|  21 |  62 |  27 |  90 |  59 |
|  63 |  26 |  40 |  26 |  72 |
|  36 |  11 |  68 |  67 |  29 |
B =
|  82 |  30 |  62 |
|  23 |  67 |  35 |
|  29 |   2 |  22 |
|  58 |  69 |  67 |
|  93 |  56 |  11 |
C = A * B
| 14638 | 14352 | 10745 |
| 15128 | 9538 | 8230 |
| 11760 | 8200 | 8921 |
```

Question 13 (1)

Écrire la(les) fonction(s) suivante(s):

```
void matrix_line_sum(struct matrix *A, int sum_line[]);
```

`matrix_line_sum(A, sum_line)` calcule la somme de chaque ligne de la matrice et stocke le résultat dans le tableau `sum_line` déjà aloué mais pas initialisé.
Les matrices sont représentée par le type suivant :

```
struct matrix {
  size_t lines, cols;
  int   *data;
};
```

De manière évidente, `lines` représente le nombre de lignes de la matrice et `cols` son nombre de colonnes. Le champ `data` est un pointeur sur une zone mémoire contenant `lines * cols` entiers correspondant aux cases de la matrice stockées en mode *ligne d'abord*.
Pour accéder à la case (i, j) de la matrice `A`, on utilisera le décallage classique : `A->data[i * A->cols + j]`.

Exemple 13.1:

```
shell> ./question13 0 3 5
Random tests:
A =
|  83 |  86 |  77 |  15 |  93 |
|  35 |  86 |  92 |  49 |  21 |
|  62 |  27 |  90 |  59 |  63 |
matrix_line_sum(A, line_sum)
| 354 | 283 | 301 |
```

Question 14 (1)

Écrire la(les) fonction(s) suivante(s):

```
size_t tree_size(struct tree *tree);
```

`tree_size(tree)` calcule la taille de l'arbre binaire `tree`.
Le type représentant les arbres est le suivant :

```
struct list {
  struct list *next;
};
```

Exemple 14.1:

```
shell> ./question14 0 3
Random Tests:
tree =
```

```
  <0
    <1
      <2
        <3>
        <4>
      >
      <5
        <6>
        <7>
      >
    >
    <8
      <9
        <10>
        <11>
      >
      <12
        <13>
        <14>
      >
    >
  >
tree_size(tree) = 15 - [OK]
```

Question 15 (1)

Écrire la(les) fonction(s) suivante(s):

```
struct tree* bst_find(struct tree* tree, int key);
```

`bst_find(tree)` cherche la clef `key` dans l'arbre binaire de recherche `tree`. La fonction renverra le pointeur sur le nœud contenant la clef ou `NULL` si elle ne trouve pas la clef.

Le type représentant les arbres est le suivant :

```
struct tree{
  struct tree *left, *right;
  int key;
};
```

On rappelle que dans un arbre binaire de recherche, les clefs sont classées : la clef de la racine (locale) est supérieure à l'ensemble des clefs de son sous-arbre gauche et inférieure à l'ensemble des clefs de son sous-arbre de droite.

Exemple 15.1:

```
shell> ./question15 0 3
Random Tests:
tree =
  <29
    <14
      <5
        <2>
        <10>
      >
      <21
        <17>
        <24>
      >
    >
    <42
      <35
        <32>
        <38>
      >
      <50
        <46>
        <57>
      >
    >
  >
bst_find(tree, 14) = found 0x61300000de98 (key: 14)  [OK]
bst_find(tree, 124) = not found
bst_find(tree, 2) = found 0x61300000dec8 (key: 2)
```

Question 16 (3)

Écrire la(les) fonction(s) suivante(s):

```
size_t newline_counter(int fd);
```

`newline_counter(fd)` lit le contenu du flux du *file descriptor* `fd` (un fichier ou l'entrée standard) et compte le nombre d'occurrence du caractère *newline* (`'\n'`).

Vous utiliserez l'appel système `read(2)` pour lire dans `fd`. On rappelle que vous devez regarder la valeur de retour de `read(2)` : si celle-ci est 0 c'est que vous avez atteint la fin de fichier, si elle est positive elle correspond au nombre d'octets lus et sinon si elle vaut -1, une erreur c'est produit. Pour la gestion d'erreur, vous vous baserez sur l'extrait de code suivant dans lequel `r` contient la valeur de retour de `read(2)` (on suppose également que vous êtes dans une boucle de lecture) :

```c
    // Something goes wrong
    if (r == -1) {
      // Not a real error, we should continue reading
      if (errno == EINTR)
        continue;
      // for all other cases, die
      err(3, "read failed");
    }
```

Exemple 16.1:

```
shell> ./question16 skel.h
newline count: 16
shell> echo -n "Some text\nwith 2 newline\n" | ./question16 -
newline count: 2
```

Question 17 (4)

Écrire la(les) fonction(s) suivante(s):

```c
void get_status(char *cmd, char **args, struct status *status);
```

`get_status(cmd, args, status)` execute la commande `cmd` avec le tableau d'arguments `args` et récupère son status. Les informations extraite du status sont sauvées dans la structure `status`.

La structure `status` est décrite par le type suivant :

```c
struct status {
  int exited;
  int exit_status, kill_signal;
};
```

Le champ `exited` est à vrai (différent de 0) si la commande a terminé normalement et faux sinon. Le champ `exit_status` contient le code de retour de la commande en cas de terminaison normale et le champ `kill_signal` contient le numéro du signal en cas de terminaison par signal.

On rappelle le schéma d'execution d'une commande :

```
pid = fork() // fork current process and get the pid
if pid is -1 die using err(3, "can't fork")
if (pid != 0): // in the parent process
  waitpid(pid, &int_status, 0) // see waitpid(2)
  use WIFEXITED/WEXITSTATUS/WTERMSIG to fill status structure
else: // in the child process
  exec the command using execvp(3)
  if something go wrong die with err(3, "execution of %s failed", cmd)
```

Read carefully manual pages of `fork(2)`, `execvp(3)` and `waitpid(2)`.

Exemple 17.1:

```
shell> ./question17 ls Makefile
Makefile
Command ls exited normaly: OK
  exit status: 0
shell> ./question17 ls foo
ls: cannot access 'foo': No such file or directory
Command ls exited normaly: OK
  exit status: 2
shell> cat seg.c
# include <stdlib.h>

int main() {
  char *s = "foo";
  s[0] = 'g';
  return 0;
}
shell> gcc -o seg seg.c
shell> ./question17 ./seg
Command ./seg exited normaly: KO
  killed by signal: 11
shell> ./question17 foo
question17: execution of foo failed: No such file or directory
Command foo exited normaly: OK
  exit status: 3
```

Remarques sur la session exam

À la fin de l'épreuve, vous devez quitter votre session en fermant l'horloge (et uniquement en fermant celle-ci.) Dans tous les cas, votre session sera également fermée à la fin du temps imparti.

Lorsque l'horloge est fermée (volontairement, ou en fin de session), la console qui vous a demandé le token vous demande votre mot de passe. **Vous devez rentrer votre mot de passe UNIX pour que le rendu ai bien lieu.**

Vous pouvez aussi rendre sans quitter : il y a une commande (dans le shell) appellée `submission` qui déclenche un rendu comme si vous quittiez (demande de mot de passe et envoie du rendu.)

Après la fin du test (dans les minutes qui suivent) vous pouvez redémarrer votre machine et vous connecter pour re-rendre (si par exemple il y a eu une erreur pendant votre premier rendu.)

Remarques sur les squelettes de questions

Pour chaque question, un embryon de code vous est fourni. Ce code correspond au strict minimum pour que la question compile et que l'appel au programme de test échoue avec une erreur identifiable. Par conséquent, vous devez supprimer du corps des fonctions à écrire, le code provoquant l'erreur, sous peine de voir votre programme échouer lors des tests (ne laissez surtout pas la ligne `REMOVE_ME(...)`.)

Exemple 1:
À titre d'exemple, si l'on vous demande la fonction C suivante :

```
int identity(int x);
```

`identity(x)` renvoi `x`.
Vous trouverez dans le fichier de question correspondant le code :

```
int identity(int x) {
  /* FIX ME */
  REMOVE_ME(x);
}
```

Que vous devrez remplacer par :

```
int identity(int x) {
  return x;
}
```

Remarques sur les programmes de test

La commande `make questionXX` produira un binaire `questionXX`. Ce binaire peut être utilisé pour tester votre réponse à la question X du sujet. Tous les binaires attendent des paramètres et affichent une aide succincte s'il sont appelés avec l'option `-help`.

Exemple 2:
Le binaire produit pour la question 1 (il s'agit d'un exemple qui ne correspond pas forcement au sujet)

```
Question 1:
./question01 graine taille
  -help   Display this list of options
  --help  Display this list of options
```

Les deux paramètres sont donc `graine` et `taille`. Les paramètres nommés `graine` (présent pour chaque question, ou presque) font tous référence à l'initialisation d'un générateur de nombre aléatoire. Dans le cas présent la commande `./question03 X Y` va initialiser le générateur de nombre aléatoire avec X (ici, Y servant de taille à la liste générée.) Pour les mêmes valeurs de X on obtiendra les mêmes données (la séquence engendrée par une graine donnée est toujours la même.)

Le reste du temps les noms des paramètres donnés dans l'aide sont explicites (par rapport au sujet.) Si vous avez une doute sur l'utilisation vous pouvez jeter un oeil sur fichier contenant les tests de la question X (`test_qXX.c`.)