EPITA S3 promo 2019
Programming - Machine Test

Marwan Burelle[*]

Friday, October 30 2015

Instructions:

You must read the whole subject and all these instructions. Every explicit instructions in the subject are mandatory. Points lost for ignoring subject rules are not open to arguments, including compiling issues or usage of directory hierarchy.

Your home directory during the test is temporary, in this directory you'll find a directory `subject` *and a directory* `submission`*. In the* `subject` *directory, you'll find a sub-directory called* `Skel`*, you have to copy the* ***content*** *of this directory in your* `submission` *directory.*

You must make regular uploads to be sure not to lose your work. In order to upload your work, simply call the command `submission`.

Here is given as example, commands to perfom the needed copy from `subject/Skel` *directory to* `submission` *directory:*

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

In this directory you'll find: a `Makefile` *offering targets to compile your code, some annex files, a file for each questions named* `questionXX.c`*. Only question files can be modified, all other files wil be replaced by the original one during the automatic correction.*

The `Makefile` *offers a target building a test program for each question. This test program will perform all the interraction part (input and output) and call your (or yours) function(s) with the correct expected parameters. In order to target the build of these test programs, you need to issue the command (for question number XX):* `make questionXX`.

During automatic correction, this `Makefile` *will be used and thus the question XX will be evaluated only if* `make questionXX` *succeed. Of course,* ***the grade will depends on the correctness of your answer.***

Scale information are only indicative and may be changed later.

At the end of the document, you'l find the extra sections providing advices about test programs.

There are 24 points and 13 questions in this test.

[*]marwan.burelle@lse.epita.fr

Question 1 (1)

Write the following function(s):

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

`fact(n)` compute factorial of n.

Factorial sequence is defined by:

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

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)

Write the following function(s):

```
unsigned long fibo(unsigned long n);
```

`fibo(n)` compute the rank n of the Fibonacci sequence.

The Fibonacci sequence is defined by:

$$\texttt{fibo}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \texttt{fibo}(n-1) + texttt fibo(n-2) & \text{otherwise} \end{cases}$$

Exemple 2.1:

```
shell> ./question02 0 5
Fixed tests:
fibo( 0) = 0
fibo( 1) = 1
fibo( 2) = 1
fibo( 3) = 2
fibo( 4) = 3
fibo( 5) = 5
Random tests:
fibo(33) = 3524578
fibo(36) = 14930352
fibo(27) = 196418
fibo(15) = 610
fibo(43) = 433494437
```

Question 3 (1)

Write the following function(s):

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

`my_intsqrt(n)` compute the integer part of the square root of n.

The integer part of the square root is the (integer) solution x of the inequation:

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

In order to solve this problem, we'll use the Heron method (variation of the Newton method.) Let x be an upper approximation of the square root (thus bigger than the root but smaller than n, we can take n as a initial approximate value.) We compute the next approximation as the arithmetic mean between x and n/x et we continue while x is bigger than n/x (*i.e.* x is bigger than the expected root.)

Exemple 3.1:

```
shell> ./question03 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 4 (1)

Write the following function(s):

```
void swap(int *a, int *b);
```

`swap(a,b)` exchange values pointed by a et b.

Exemple 4.1:

```
shell> ./question04 0 5
Testing swap:
        Before swap: a = 1804289383 | b =  846930886
        After  swap: a =  846930886 | b = 1804289383    [OK]

        Before swap: a = 1681692777 | b = 1714636915
        After  swap: a = 1714636915 | b = 1681692777    [OK]

        Before swap: a = 1957747793 | b =  424238335
        After  swap: a =  424238335 | b = 1957747793    [OK]

        Before swap: a =  719885386 | b = 1649760492
        After  swap: a = 1649760492 | b =  719885386    [OK]

        Before swap: a =  596516649 | b = 1189641421
        After  swap: a = 1189641421 | b =  596516649    [OK]
```

Question 5 (2)

Write the following function(s):

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

`array_sum(begin, end)` computes the sum of the cells in the array betwee `begin` (included) and `end` (excluded.) If `end - begin = 0`, the function will return 0.

Exemple 5.1:
```
shell> ./question05 0 5
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |
  array_sum = 15 [OK]
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 |
  array_sum = 3754
```

Question 6 (2)

Write the following function(s):

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

`array_min(begin, end)` finds the minimum value of the cells in the array betwee `begin` (included) and `end` (excluded.) The function is only defined if `end - begin > 0`.

Exemple 6.1:
```
shell> ./question06 0 5
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |
  array_min = 1 [OK]
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 |
  array_min = 383
```

Question 7 (3)

Write the following function(s):

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

`avg_partition(begin, end)` groups, in the first part of the array, values lower than the average value of the array and groups upper values in the second part of the array. The function returns the address of the first cell containing (after grouping) a value greater than the average.

The partition is similar to the quick sort partition, the only difference is the choice of the pivot value that you obtain by computing the average of the whole array. Here is a pseud-code algorithm for the partitionning:

```
avg_partition(tab, left, right):
  a = average of tab
  pivot = left
  for i = left to right - 1 do:
    if tab[i] <= a:
      swap tab[i] and tab[pivot]
      pivot = pivot + 1
    end if
  done
  return pivot
```

Exemple 7.1:

```
shell> ./question07 0 8
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |
  run avg_partition
  lower half:
    |   1 |   2 |   3 |   4 |
  upper half:
    |   5 |   6 |   7 |   8 |
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 | 335 | 386 | 492 |
  run avg_partition
  lower half:
    | 383 | 335 | 386 | 492 |
  upper half:
    | 793 | 886 | 777 | 915 |
```

Question 8 (3)

Write the following function(s):

```c
void array_merge(int *dst, int *a1, int *a2, int *e1, int *e2);
```

`array_merge(dst, a1, a2, e1, e2)` merges the two sorted arrays `a1` and `a2` in the array `dst`. `e1` (respectively `e2`) is the end pointer (excluded) of the array `a1` (respectively `a2`). The area pointed to by `dst` is supposed to be sufficient in order to contain the content of both arrays (the end of `dst` is at address `dst + e1 - a1 + e2 - a2`.)

There's no constraint of size betwee `a1` and `a2`.

Of course, the result of merging (in `dst`) must be sorted.

Exemple 8.1:

```
shell> ./question08 0 7
Fixed tests:
  array1:
    |   1 |   2 |   3 |
  array2:
    |   4 |   5 |   6 |   7 |
  array_merge(dst, array1, array2, ...)
  dst:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |

Fixed tests (2):
  array1:
    |   1 |   3 |   5 |   7 |
  array2:
    |   2 |   4 |   6 |
  array_merge(dst, array1, array2, ...)
  dst:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |

Random tests:
  array1:
    |  10 |  28 |  44 |
  array2:
    |  19 |  26 |  39 |  49 |
  array_merge(dst, array1, array2, ...)
  dst:
    |  10 |  19 |  26 |  28 |  39 |  44 |  49 |
```

Question 9 (4)

Write the following function(s):

```c
void array_rot(int array[], size_t len, size_t offset);
```

`array_rot(array, len, offset)` *rotate* of `offset` cells the content of the array named `array` (of length `len`).

Rotating an array works by considering the array as a *circular* one and shifting cells to the right. Since the array is circular, for a rotation of 1, the last cell take the place of the first one (and the first one take the place of the second . . .)

You can implement an naive version of the rotation of `offset` cells, as repeating `offset` times a rotation of 1 cell. But, a more efficient version works by first saving (copying) the last `offset` cells to a temporary array, then shift by `offset` cells to the right the original array and finally copying back the saved cells at the beginning of the array. The following pseudo-code describes these steps:

```
array_rot(array, len, offset):
  tmp = malloc(offset * sizeof (int))
  copy offset cells from array[len - offset] to tmp
  shift array by offset cells to the right
  copy offset cells from tmp[0] to array
  free(tmp)
```

For this version, you need to allocate (using `malloc(3)`) a temporary array (with enough rooms, thus `offset` integers.) Of course, you'll need to free the temporary array at the end (using `free(3)`.)

Exemple 9.1:

```
shell> ./question09 0 8
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |
  array_rot(array, 8, 1)
  array:
    |   8 |   1 |   2 |   3 |   4 |   5 |   6 |   7 |
  array_rot(array, 8, 4)
  array:
    |   4 |   5 |   6 |   7 |   8 |   1 |   2 |   3 |
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 | 335 | 386 | 492 |
  array_rot(array, 8, 2)
  array:
    | 386 | 492 | 383 | 886 | 777 | 915 | 793 | 335 |
```

Question 10 (3)

Write the following function(s):

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

`insert_sort(begin, end)` sorts the array betwee `begin` (included) and `end` (excluded.) We'll use the insertion sort, here is the algorithm:

```
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 10.1:

```
shell> ./question10 0 5
./question10 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 11 (1)

Write the following function(s):

```
str_toupper(char *s);
```

`str_toupper(s)` translates all lower case letters of the string s (terminated by `'\0'`) into upper case and leaves unchanged all other characters.

Remember that characters in C are just integer. Let c be a `char` variable containing a lower case letter, then c verifies: `'a' <= c && c <= 'z'`. Similarly, if c contains an upper case letter, then: `'A' <= c && c <= 'Z'`.

Exemple 11.1:
```
shell> ./question11 0 26
Fixed tests:
  s = "abcdefghijklmnopqrstuvwxyz"
  str_to_upper(s)
  s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Random tests:
  s = "n{6\Pavw[:mO4=ZvMD^bvU;e'R"
  str_to_upper(s)
  s = "N{6\PAVW[:MO4=ZVMD^BVU;E'R"
```

Question 12 (1)

Write the following function(s):

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

`mystrlen(s)` function calculates the length of the string *s*, excluding the terminating null byte (`'\0'`). The input pointer is not `NULL`. You must respect the expected behavior of the function `strlen(3)`.

Exemple 12.1:
```
shell> ./question12 0 5
s = "n{6\P"
mystrlen(s) = 5 -- check: [OK]
```

Question 13 (1)

Write the following function(s):

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

`mystrncpy(dst,src,len)` : function copies at most `len` bytes of the string pointed to by `src` to the buffer pointed to by `dst`. The strings may not overlap, and the destination string `dst` must be large enough to receive the copy. If there is no null byte among the first `len` bytes of `src`, the string placed in `dst` will not be null-terminated. If the length of `src` is less than `len`, `mystrncpy()` writes additional null bytes to `dst` to ensure that a total of `len` bytes are written.

Note: `mystrncpy()` **always** writes exactly `len` bytes, whatever is the length of `src`.

You should read carefully the manual page of `strncpy(3)` which provides a complete description of the expected function.

Exemple 13.1:

```
shell> ./question13 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]
```

About The Test Session

Once the test is over, you must leave your session by closing the clock (that's the only way.) Note that when the test is over, your session will close directly.

When the session closed, you'll be prompted for your password (the one used to login.) This will end the test (your *submission* directory will be archived and sent to the collecting server.) **You must not shutdown the computer before the completion of this final step, otherwise your work will be lost.**

You can send intermediary versions of your test by using the shell command `submission`. It is strongly advised that you do so to prevent data lost before the end of the test.

Even after the end of the test (in the few minutes following the test, of course), you can restart your computer to eventually re-send your work (this may be required sometimes if something goes wrong during the final step.)

About Questions Skel

For every question, a skeleton of code is provided. This code is the minimal requirement for the compilation of the file *w.r.t.* the test program. The content of the skeleton will also induce a failure at execution time and thus you must remove the the body of the function(s). In C, be sure to remove (or comment) the `REMOVE_ME()` line of code: if it's still in the file, it will probably be executed anyway.

Exemple 1:
For example, if you're asked for the following function:

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

`identity` x returns x.
You'll find the following skeleton:

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

Your answer will look like:

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

About test programs

When invoked with `make questionXX`, `make` will build a binary program named `questionXX`. This program can be used to test your answer to the question X in this subject. All binary wait for parameters et display a small help when run with `-help`.

Exemple 2:
For example, the program for question 1 (this is an example and may not corresponds to the actual question 1) will display:

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

The two parameters are thus: `graine` (seed) and `taille` (size). These parameters are present in most question: the seed is used to initilize the random number generator (for a given seed, the generator will produce the same sequence of number) and the size can be either the size of generated data (for list or strings . . .) or the number of tested . . .

If you need more detail, read the `test_qXX.c` files.