

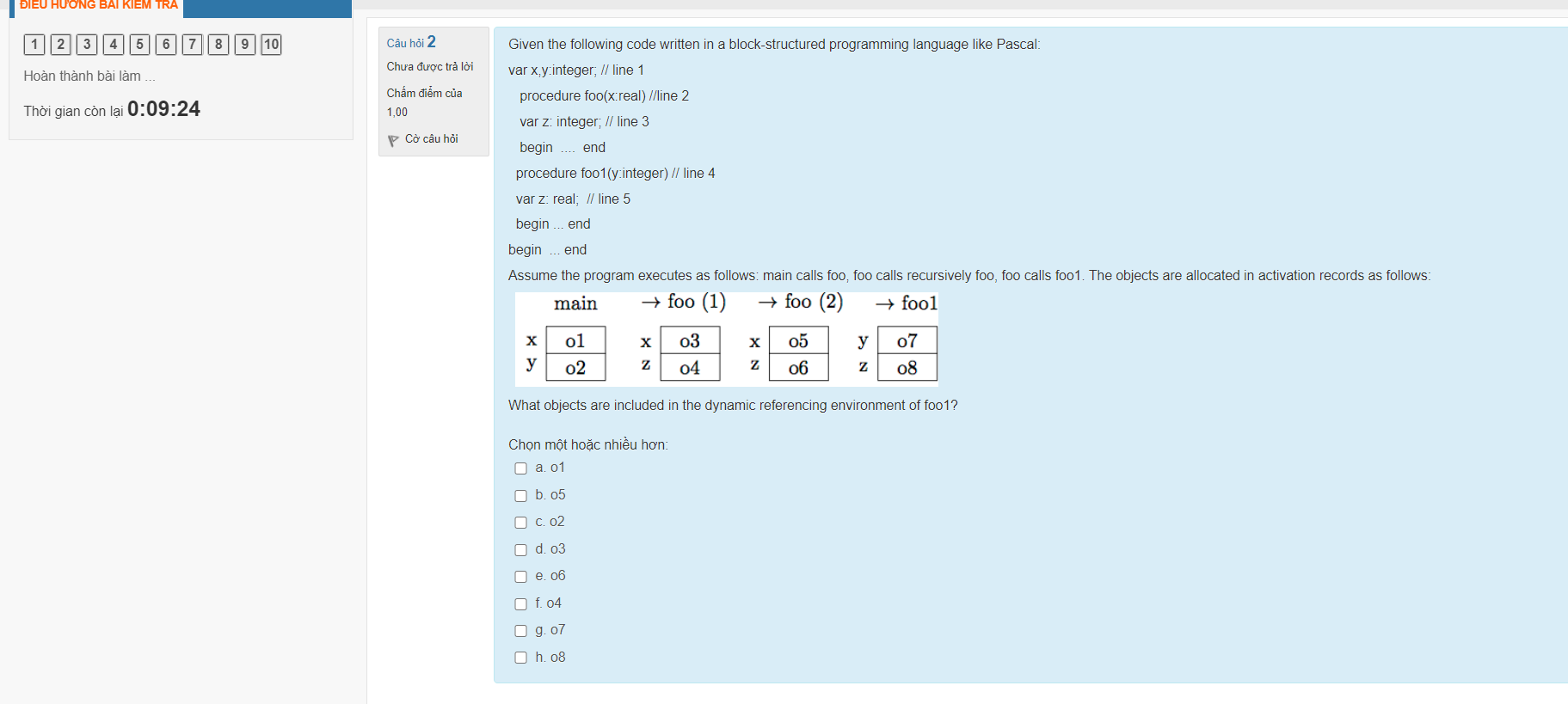
m => Stack, if we’re talking about the pointer itself. The address it points to is unallocated.

x => Probably CPU registers, not really on the stack.

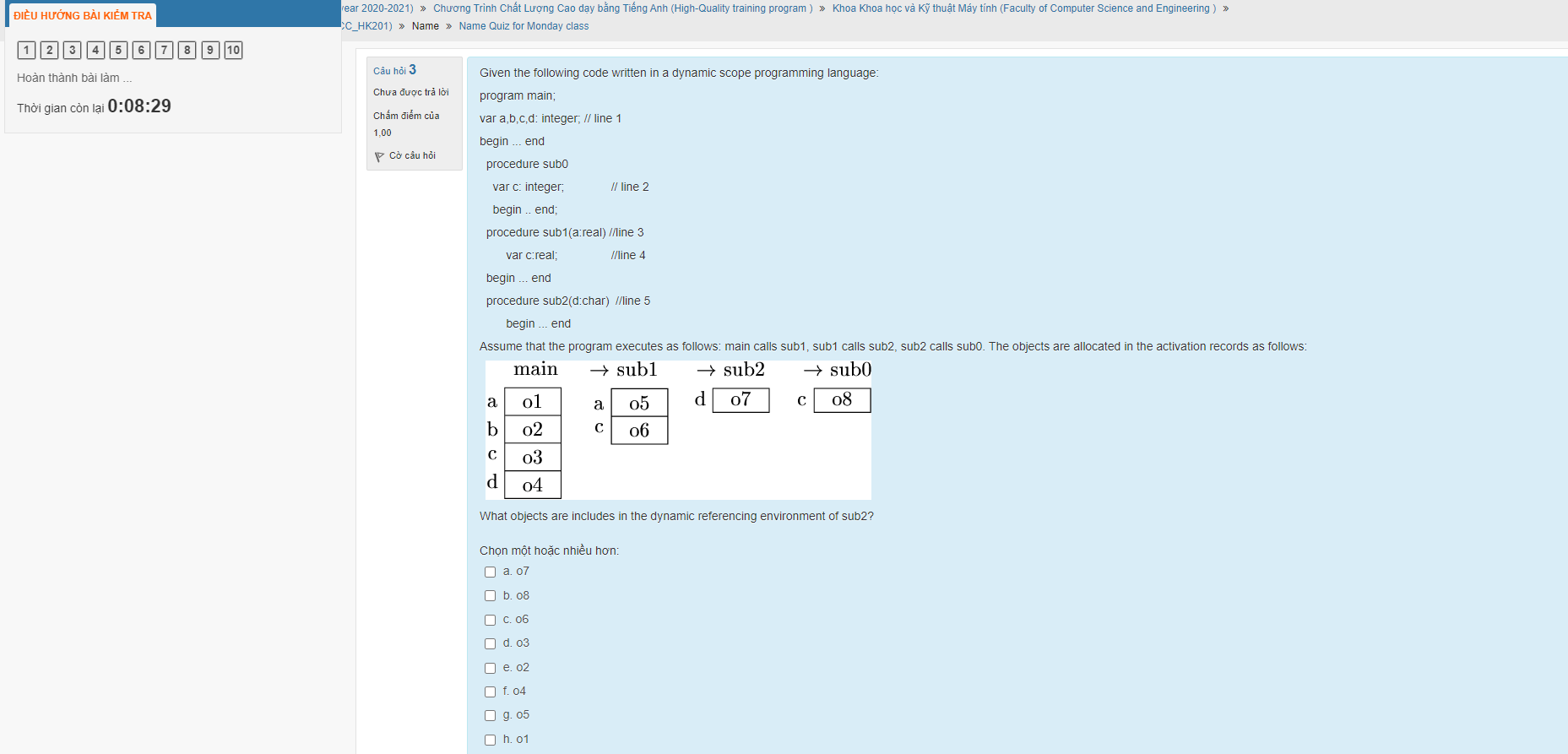
y => Data segment, since it’s static.

z => Stack (the pointer itself).

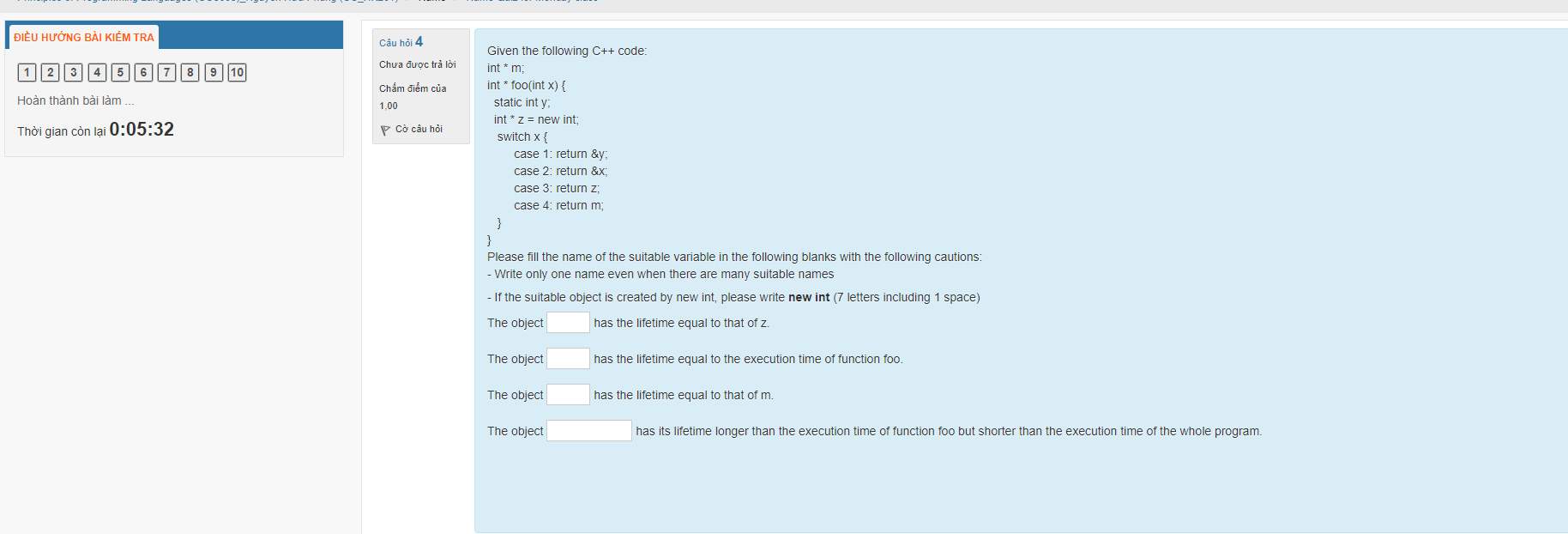
The object created by **new int** => Heap.



x (o5), y(o7), z(o8) => Tick o5, o7, o8.



a(o5), b(o2), c(o6) and d(o7) =? Tick o2, o5, o6 and o7.

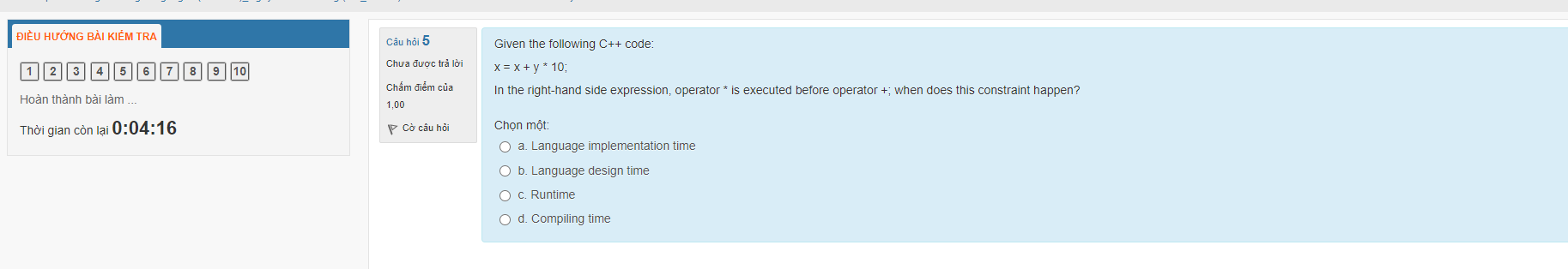


foo (the function itself)

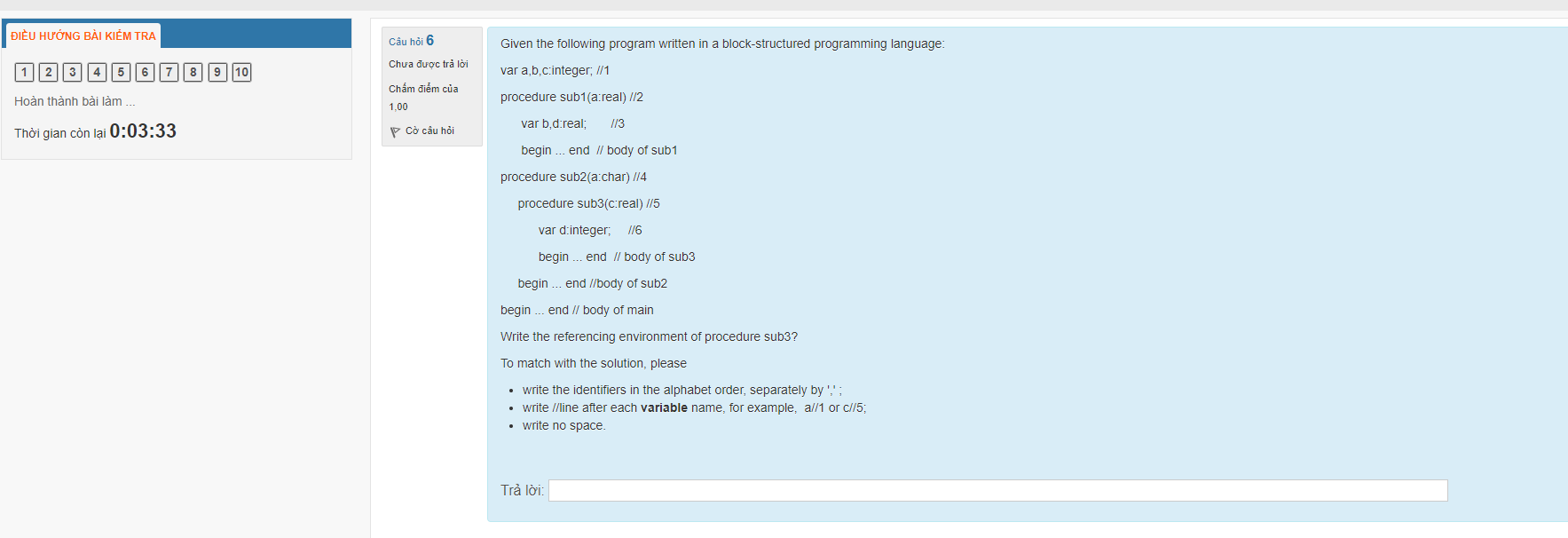
z (the pointer – it is freed from the stack after *foo* concludes)

y (static variables always live for the entire program)

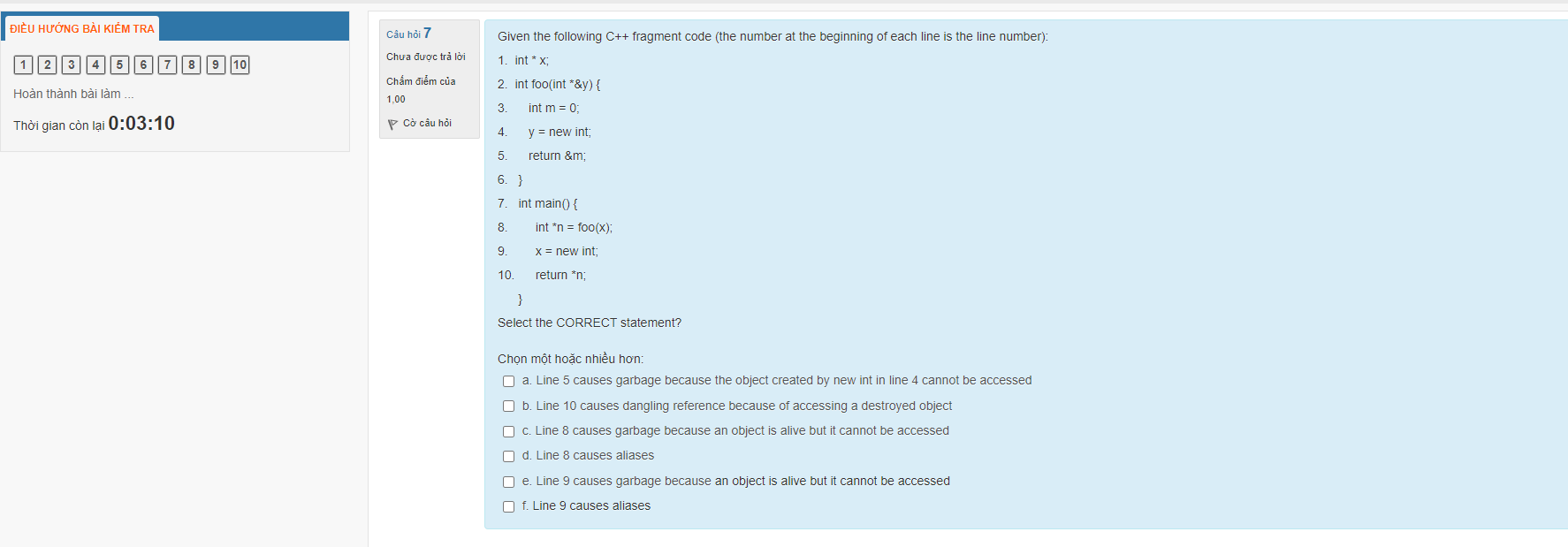
new int (it outlives *foo* and becomes a memory leak)



B, since it depends on the design of the language. Smalltalk for example is designed such that all binary operators have the same precedence, i.e. that C code written in Smalltalk would behave differently.



a//4,b//1,c//5,d//6,sub1,sub2,sub3



B (the returned value of foo is the address of local variable *m*, which is freed after *foo* concludes, and the return statement tries to dereference it)

D (\*n is now an alias to foo(x)’s return value – although that value is technically freed)

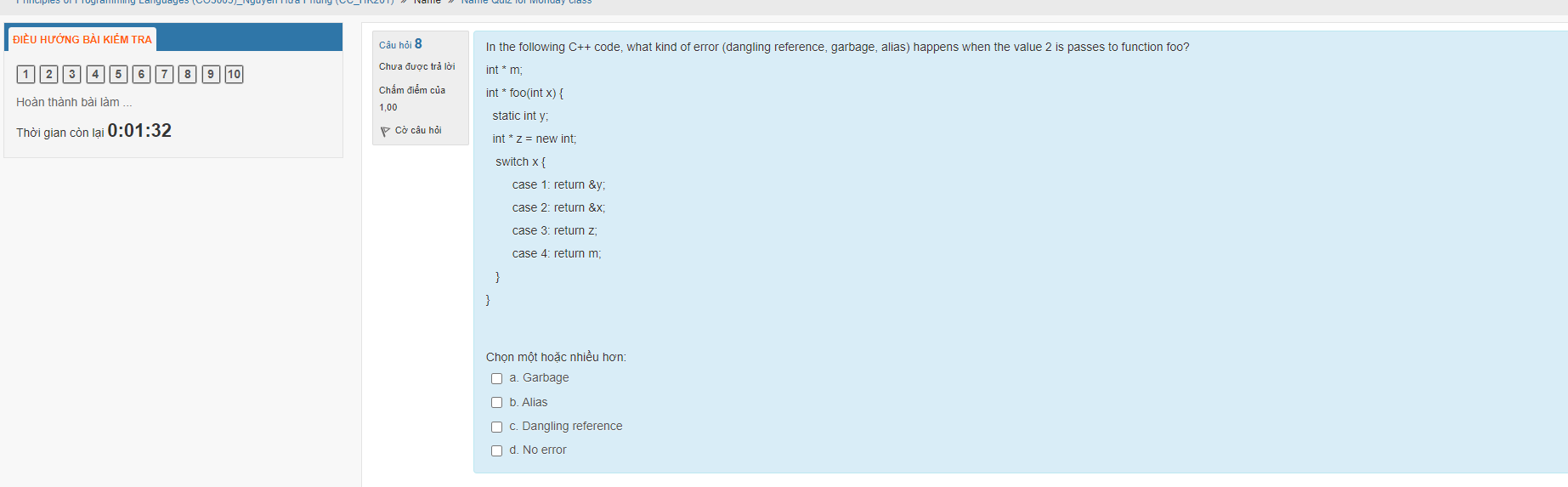
Unsure: E (x is available for us to access that new object – although the code chose not to before ending)

The rest are wrong:

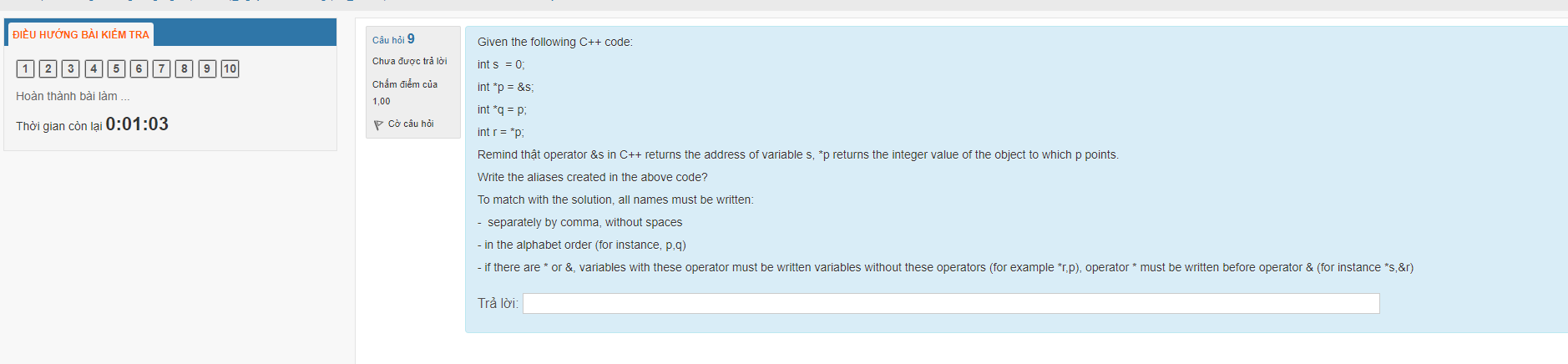
A: Although the second part is correct (the object in line 4 cannot be accessed), that constitutes a dangling reference and not garbage.

C: Line 8 creates a dangling reference (to the former local variable *m* inside *foo*), which is the opposite of garbage.

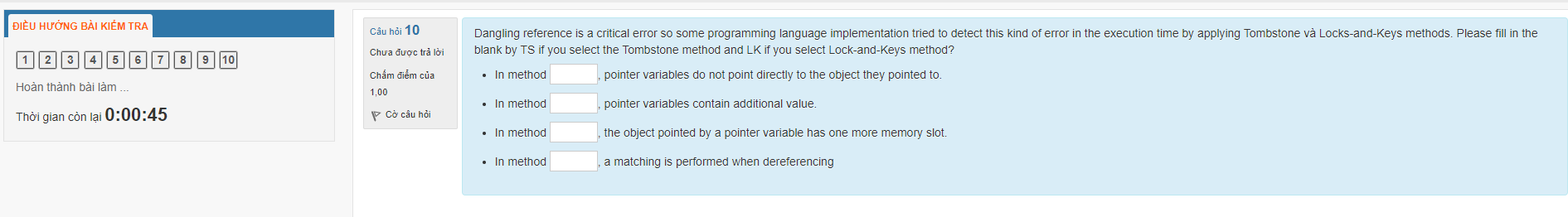
F: Line 9 does not cause an alias, since it simply allocates memory for a known (and only) variable name (x).



Dangling reference, since *x* is passed by value and as such is a copy that only lives as long as *foo* itself.



\*p,\*q,s



TS

LK

LK

LK