15-122: Principles of Imperative Computation

Recitation Week 4 Solutions

Nivedita Chopra, Rob Simmons

Checkpoint 0

Write a function to reverse a queue, using only the functions from the stack and queue interfaces.

```
1 void reverse(queue Q) {
2
3     stack S = stack_new();          // Hint : Allocate a
4                                            // temporary data structure
5     while(!queue_empty(Q)        ) {
6
7         string temp = deq(Q);
8
9         push(S, temp);
10    }
11    while(!stack_empty(S)        ) {
12
13        string temp = pop(S);
14
15        enq(Q, temp);
16    }
17 }
```

Checkpoint 1

Write a *recursive* function to count the size of a stack.

```
1 int size(stack S) {
2
3     if (stack_empty(S)) return 0;
4
5     string x = pop(S);
6
7     int i = size(S);
8
9     i++;
10
11    push(S, x);
12
13    return i;
14
15    ______________________________
16 }
```

Checkpoint 2

Why couldn't this stack size implementation be used in contracts in C0?

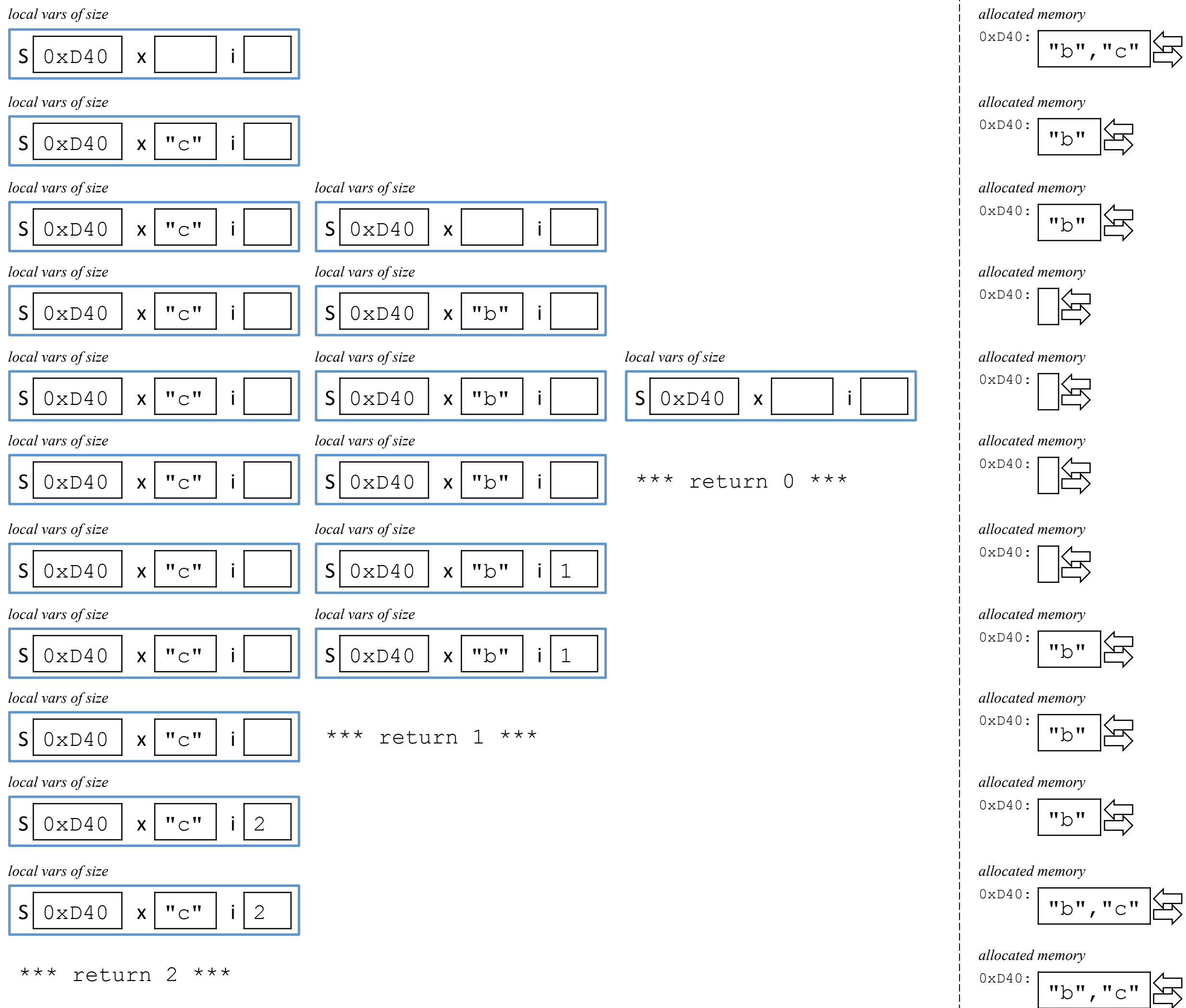
Solution: C0 has checks in place to make sure that any functions that are called from the assertion language (`//@requires`, `//@ensures`, `//@loop_invariant`, or `//@assert`) are are *pure* – that is, they don't manipulate memory. This is to ensure that code running with contracts will always return the same answer as code running without contracts. (It's possible to turn off this check with the –no-purity-check option.)

Because pushing and popping from the stack modifies allocated memory (we haven't seen how, yet, but we know, even with our abstract picture of stacks, that it must), the size function we wrote above isn't pure.

Checkpoint 3

The above example works because function calls use a data structure that is like a stack. Step by step, trace out operationally the state of the computer's memory when it calculates the size of a stack with two strings "b" and "c", taking account of the fact that each recursive call gets its own copy of the assignable variables.

Solution:



Checkpoint 4

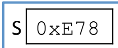
In the same fashion, trace out what happens operationally in this broken reversal function, starting with the code in `main()`.

```
1  void reverse(stack S) {
2     string x;
3     stack R = stack_new();
4
5     while (!stack_empty(S)) {
6        x = pop(S);
7        push(R, x);
8     }
9
10    S = R;
11 }
12
13 int main() {
14    stack S = stack_new();
15    push(S, "foo");
16    reverse(S);
17    println(pop(S));
18    return 0;
19 }
```

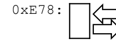
local vars of main

allocated memory

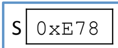
local vars of main



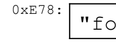
allocated memory



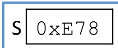
local vars of main



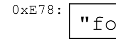
allocated memory



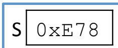
local vars of main *local vars of reverse*

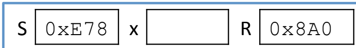


allocated memory

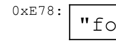


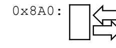
local vars of main *local vars of reverse*



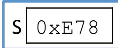


allocated memory

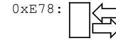


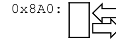


local vars of main *local vars of reverse*

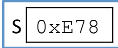


allocated memory

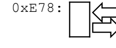


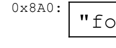


local vars of main *local vars of reverse*

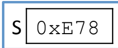


allocated memory

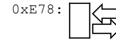


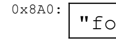


local vars of main *local vars of reverse*

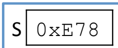


allocated memory



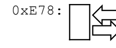


local vars of main

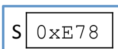


`*** return ***`

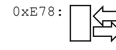
allocated memory



local vars of main



allocated memory



`*** Safety violation! The stack S is empty.***`