Pointer manipulations

Pointer manipulations are counterintuitive at first, but with just a little bit of practice, they will become second nature. A key insight is the following:

*When you set a pointer equal to another pointer, you make the first pointer point to where the second pointer points.*

Consider the following memory diagram, where `a` has type `int*` and `b` has type `int**`:

```
Local mem.          Allocated mem.
                      122

a [0] -> [122]  

b [0] -> [0]  
```

Draw the memory diagram after executing the code line

```
*b = a;
```
A wild struct appears

Suppose we have the following in a file:

```c
struct X {
    int    a;
    struct Y* b;
};

struct Y {
    int*   a;
    int    b;
    struct X* c;
};

void f(struct X* x, struct Y* y) {
    x->b = y;
    y->c = x;
    y->c->a = 15
    int** d = alloc(int*);
    *d = alloc(int);
    x->b->a = *d;
    *(y->a) = x->a * 8 + 2;
    x->b->b = 1000 * x->a + **d;
    x = NULL;
    y->c = NULL;
    return;
}

int main() {
    struct X* foo = alloc(struct X);
    struct Y* bar = alloc(struct Y);
    return 0;
}
```

Checkpoint 0

Fill out the state of the memory. What’s the value of `bar->b`? (For your own sanity, draw a picture!)
Stack and queue interfaces

Here’s the stack interface discussed in lecture. It exposes the type stack_t and four functions:

```c
// typedef ______* stack_t; /* Abstract type of stacks */

bool stack_empty(stack_t S) /* Check if stack S is empty, 0(1) */
/*@requires S != NULL; @*/ ;

stack_t stack_new() /* Create a new empty stack, 0(1) */
/*@requires \result != NULL; @*/;
/*@requires stack_empty(\result); @*/ ;

void push(stack_t S, string x) /* Add item x at the top of stack S, 0(1) */
/*@requires S != NULL; @*/;
/*@requires !stack_empty(S); @*/ ;

string pop(stack_t S) /* Remove and return the top of stack S, 0(1) */
/*@requires S != NULL; @*/;
/*@requires !stack_empty(S); @*/ ;
```

The queue interface exposes the type queue_t and four similar functions:

```c
// typedef ______* queue_t; /* Abstract type of queues */

bool queue_empty(queue_t Q) /* Check if queue Q is empty, 0(1) */
/*@requires Q != NULL; @*/ ;

queue_t queue_new() /* Create a new empty queue, 0(1) */
/*@requires \result != NULL; @*/;
/*@requires queue_empty(\result); @*/ ;

void enq(queue_t Q, string e) /* Add item e at the back of queue Q, 0(1) */
/*@requires Q != NULL; @*/;
/*@requires !queue_empty(Q); @*/ ;

string deq(queue_t Q) /* Remove and return the front of queue Q, 0(1) */
/*@requires Q != NULL; @*/;
/*@requires !queue_empty(Q); @*/ ;
```
Checkpoint 1
Write a function to reverse a queue using only functions from the stack and queue interfaces.

```c
void reverse(queue_t Q)
//@
{ // create temp data structure
    while ( ) {
        //
    }
    while ( ) {
        //
    }
}
```

Checkpoint 2
Write a recursive function to count the size of a stack. You may not destroy the stack in the process — the stack’s elements (and order) must be the same before and after calling this function. Assume the stack contains elements of type string.

```c
int size(stack_t S)
//@
{ //
    //
    //
    //
    //
    //
}
```

Checkpoint 3
Why couldn’t this function be used in contracts in C0? Hint: Contracts in C0 can’t have side effects.