CS330 – Spring 2016
Karina Bercan

Lab 5: I'll be C-ing you

1a) The push function takes an int and a pointer to a pointer. Why can't it just take a regular pointer? (You are encouraged to write a version of the code that takes a regular pointer in order to test this out).

We send push() *top* which is a pointer to a node struct (memory location of the node). However, we don't just send *top*, we send *&top* which is a pointer to *top*, meaning the value we are sending to push() is the memory location of *top*. A regular pointer can't be sent because it wouldn't change the value of the nodes, since structs are passed by value. In order to commit changes, memory addresses must be accessed, not just the values changed.

1b) On the other hand, the printStack function does only take a pointer, not a pointer to a pointer. Why is that okay?

Printing the stack doesn't change the data structure, so it doesn't matter if memory locations are referenced or not; accessing the values is enough.

2) Why is the while loop in the printStack function written the way it is?

The while loop runs while the value of *next* in the node that was sent into printStack is not NULL (while the address sent in is a node and has a next node). The loop prints the int held in the current node and the address of the next node, then it sets the current node to the next node to alter the conditions that keeps us in the loop. It is written this way so that the two values held in the current node's attributes are printed, until there are no more nodes in the stack.

3) Write a function that returns the number of nodes in the stack.

```
int stackSize(struct node* n){
   int size = 0;
   if (n==NULL){
      // says the list is empty
      printf("\n\nThe list is empty!");
   }
   // if the address holds a node
   else{
      size++;
      // while the value of next in the node that was sent in is not NULL
      // meaning, while the address sent in is a node and has a next node
      while(n->next != NULL){
         // increment the count
         size++;
         // set the current node to the next node
         // this will alter the conditions that keeps us in the loop, eventually
         n = n->next;
      }
```

```
    }
    return(size);
}
```

4) There are two different push functions, one that works and one that's broken. Why does the one that's broken not work? Why does the one that works, work? (Hint: You will need to do some research on what malloc does, and think about scope and memory allocation).

malloc() allocates memory space that is big enough for the data type you give it. In the working push() function, `struct node* n = (struct node*) malloc(sizeof(struct node))` **creates a pointer to a node and calls it *n*. malloc() allocates memory space of the size needed for a node, and the code contained in parentheses sets the data type of the address to a pointer to a node. In pushBroken(), no memory space is allocated explicitly, so the scope of the node created in it is limited to that function; it is not actually visible to the stack outside of the function call.**

5) Write a pop function, test to make sure it works. What input parameters does it take, and how does it work?

My pop function does not work, but this is what I did:
The input parameter is a pointer to a node and it returns a pointer to a node. If the stack is empty, it returns a null value. If the stack is not empty, it saves the top node in another node pointer, and it prints the value in that node. Then, it sets the top node sent into pop() to the next node. It finally returns the top node.