Assignment 3 Design - Jack Vento

Following the organization and function prototypes supplied, I attacked this problem first iteratively to develop a real understanding of what is happening with the Tower of Hanoi. Then, I developed the recursive solution, which was significantly easier and largely self-solving.

Stack.c

I didn't do anything particularly interesting for any of these definitions. Their implementations largely follow the lecture slides with a few more if statements checking the validitity of allocated memory (specifically in regards to the dynamic array).

Create

- Allocate stack on the heap and check that creation was successful. If not, return null. Continue otherwise.
- Assign capacity to the supplied size.
- Assign top to 0.
- Allocate an array (items) of supplied size on the heap and check that it succeeded.
- Return null if unsuccessful, otherwise return a ptr to the stack.

Delete

- Free and null the items array.
- Free and null the stack.

Pop

- If the stack and the top are valid, decrement the top.
- Return reference to the new top if valid, else -1.

Push

- If top = capacity, reallocate the items array to a larger capacity.
- Set top to be supplied value and post increment top.

Empty

- Return true if top = 0, else return false.

Peek

After struggling with the implementing the move function, I came back to this function and added three different return states that I could handle when trying to move disks around. Originally, it only returned the item. Now, it handles the following cases:

- -1: We are trying to access an invalid stack or stack array (malloc failed or invalid ptr);

- 0: The stack's empty. This was particularly helpful in determining which peg (aka stack) to move the disk to.
- Other integer: If the stack isn't empty, return the item below the top (s->items[top - 1]).

Tower.c

Stack

Parameters: n (number of disks), begin char (start peg), final char (target peg), helper char (other peg)

Design: After watching this animation at TOH.info, I noticed several patterns. First of all, the disks move clockwise when n is even and counterclockwise when odd. Rather than develop two different move algorithms, I decided to handle this by simply switching the pegs (stacks) using pointers. Secondly, movement in this game fundamentally works in groups; however many pegs there are, that is how many different movements there are. In the case of this problem, three pegs means three stacks which means three different types of movements. They are as follows:

1. The first disk (the smallest in the group of 3) always goes from the start to finish peg.
2. The second disk (the middle of the group) goes from start to the helper peg since it's empty or later occupied by larger disks.
3. The third disk has nowhere to go since it's the largest. So, to help facilitate that movement, we need to move a disk from the helper to the final peg.

This three move cycle ensures that the final peg is always ready to accept a new disk. That makes this problem the perfect place to use mod (%). Simply modding the disk number by 3 will get us one of three cases (see the finished algorithm for them specifically).

Finally, I also decided to both calculate the moves using the supplied $2^n - 1$ formula and through tracking a count with each individual move. Were I aiming for performance, I wouldn't bother with the count and would simply print the already calculated move integer (needed to insert all of our disks into the source stack). However, since this is an academic endeavor, I wanted to show that they were equal, so I calculated them individually and then compared them to ensure that my solution was correct.

Psuedocode:

- Print stack header
- Create stacks for begin, final, and helper using specified parameter chars.
- if n is even:
  - Swap final and helper to ensure proper movement direction (clockwise = even, counterclockwise = odd).
- Calculate number of moves (moves = $2^n - 1$)
- Push disks largest to smallest onto the begin stack (for moves -> 1).
- Following movement algorithm developed with help from the aforementioned animation:
- for i <= moves
  - if i % 3 == 0 (facilitate movement of the rest of begin's disks by moving from helper): Move disk from helper to final.
  - if i % 3 == 1 (simply move to the goal peg): Move disk from begin to final.
  - if i % 3 == 2 (move to the helper peg to get at the larger disks under the current disk): Move disk from begin to helper.

    - Increment the count for all 3 cases.
- Print the number of moves from count.

Moving The Disks (src to dest through helper function move_disks):

```
* Parameters: src (Source stack ptr), dest (Destination stack ptr)
    * Peek both pegs and store their results in local variables.
    * When src is empty (result of src peek was 0), push disk2 onto src and pop it from dest.
    * When dest is empty (result of dest peek was 0), push disk1 onto dest and pop it from src.
    * If neither are empty, move the smaller disk to the peg with the larger disk in the same way as above (push using the result of a pop from the other peg).
    * Print the movement for each case using a helper function (print_move) to avoid redundancy.
```

Printing the movement (helper print_move)

```
* Parameters: n (disk number), src (name character of source peg), dest (name character of dest peg)
    * Simple printf following the format from the lab manual and inserting our function parameters uisng %.
```

Recursion

Parameters: n (number of disks), count (pointer to integer from main to track moves), begin char (start peg), final char (target peg), helper char (other peg)

Design: Using the lessons learned from the TOH animation that I described above in the 'stack' section, designing the recursive solution was incredibly easy. The basic algorithm is: move the disks smaller (n - 1) than the largest one to the helper peg, move the largest disk to the final peg, then move the rest (n - 1) back to the final peg. This works because our recursive solution will do this for every disk, decrementing down by 1 to the base case, where the supplied disk is the smallest disk and can simply be moved to the target peg.

One key difference from the previous solution is that incrementing a count recursively is a tricky process and the result from the recursion stack can be unreliable at best. So, I passed in a pointer to an integer declared in main that would be immune from the dangers of the recursion stack. It's tracked there and incremented with each function call, ensuring reliability.

Psuedocode:

- Base case: n = 1 -> Print, increment count, and return
- Else
    - Recursion(n - 1, begin, helper, final)
    - Print moving disk message. Increment moves
    - Recursion(n - 1, helper, final, begin)

Main

Design: This is the least complicated part of the program. Similar to assignment 2, it primarily consists of a simple getopt loop that'll decide which functions to call. However, unlike assignment 2, I decided to opt for bools to track the command line arguments rather than enums since the problem allows multiple inputs. Using bools better allows multiple, simultaneous inputs (and takes less memory) than tracking several different enums, so I opted for them instead.

Psuedocode:

- Declare bools (run_stack, run_recurs) and size (n) to track each command line argument.
- Check if there are less than 1 command-line arguments supplied or no run preference, print an error and return if so.
- while there are options to parse
    - Set bool statuses for each argument.
    - Set size if n supplied
- if run_stack
    - Stack()
- if run_recurs
    - Print recursion header (can't do it in the function like the stack implementation or we'd have way too many prints)
    - Declare count to track moves
    - Recursion()
    - Print number of moves from count