Kristian Snyder

Data Structures CS2021 Section 001

Prof. Charles Zimmer, Prof. Karthik Vadambacheri

Lab 05

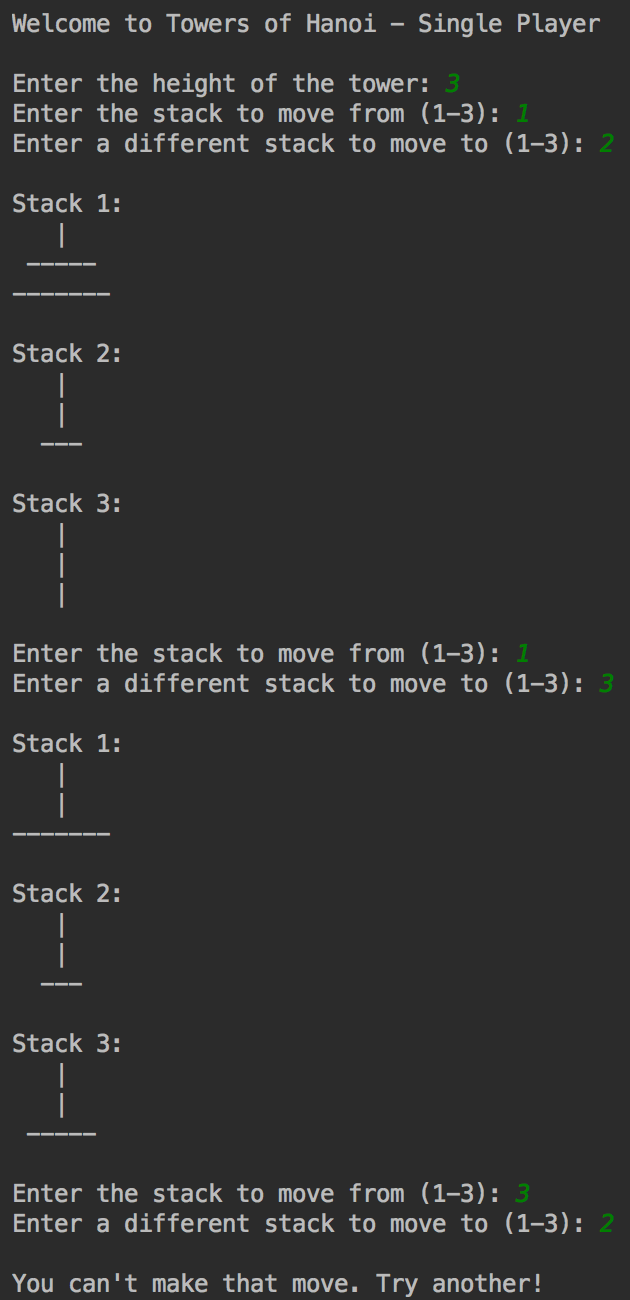
This project depended heavily on sane and easily extensible implementations of stacks and queues for success, as well as good pointer use. Without consideration of how pointers work, state in the stacks and queues, and checking to make sure neither structure is full or empty in the correct cases, the number and complexity of helper functions can bloat quickly. Additionally, the use of templating and friend functions allowed for easy code reuse when attempting to implement a two-player version of Towers, as I only had to implement a separate print function and add a more robust win check process. The underlying function of the stack used worked without issue for both cases.

Using these concepts in the workplace commonly arises when attempting to write a general class for use in many cases, both known and unknown. When writing one, especially if templated, it can be tempting to cover all possible use cases, even those that aren’t expected. However, sometimes this is not necessary or can even be detrimental to the use of the class, as users may not find a general implementation of a certain function satisfactory and are forced to override it. This results in double the time taken to write one piece of functionality. In my case, this was the print function for the stack; I found quickly that it would be much easier and better for the user if I wrote a separate function for single player and two players, as I could make assumptions about the items contained in the stack that would be impossible in a templated class.

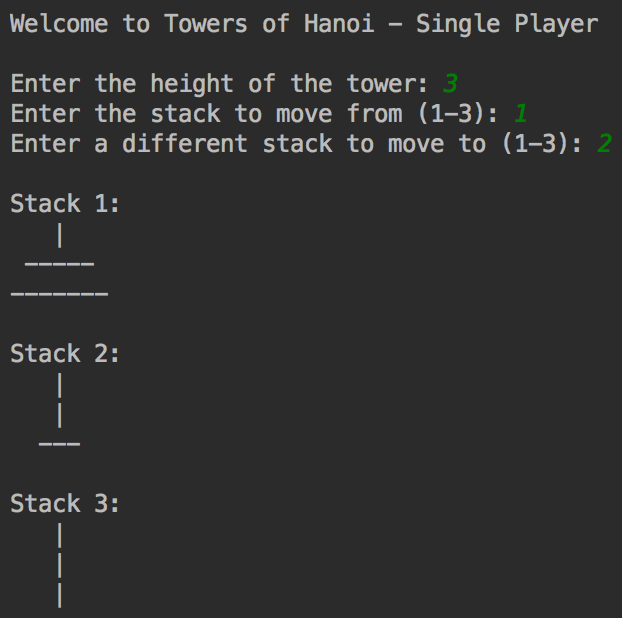
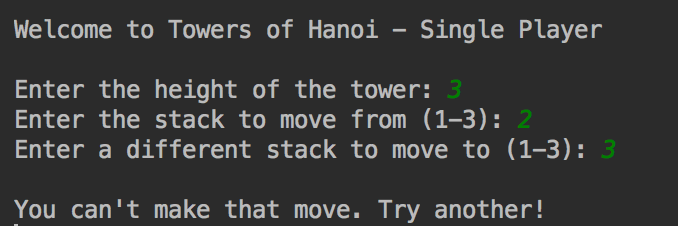
As I referenced earlier, pointer use is extremely important in this case. As C++ will not complain in many cases that pointers are used in unexpected ways, careful auditing of code using them is very important. One example I found related to comparison; C++ will happily compare two addresses in any way the programmer desires and will commonly result in difficult-to-find bugs from seemingly random logic errors. I ran into this when attempting to determine whether one disk was smaller than another. Without dereferencing the pointers inside the stack, it would simply compare the addresses of the disk and allow illegal moves. A good way to avoid this and other issues like it is to set certain policies for pointers except in carefully labeled sections of code. Not performing pointer arithmetic, comparison, and other operations like it will make it more obvious when incorrect code has been written, as both are almost always valid C++ despite very different outcomes.

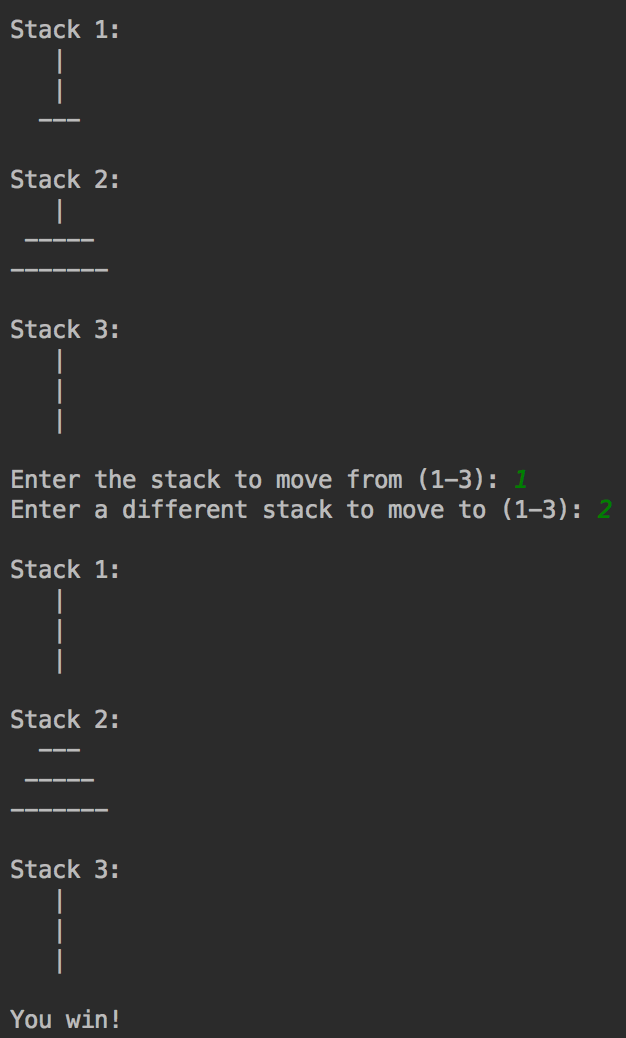
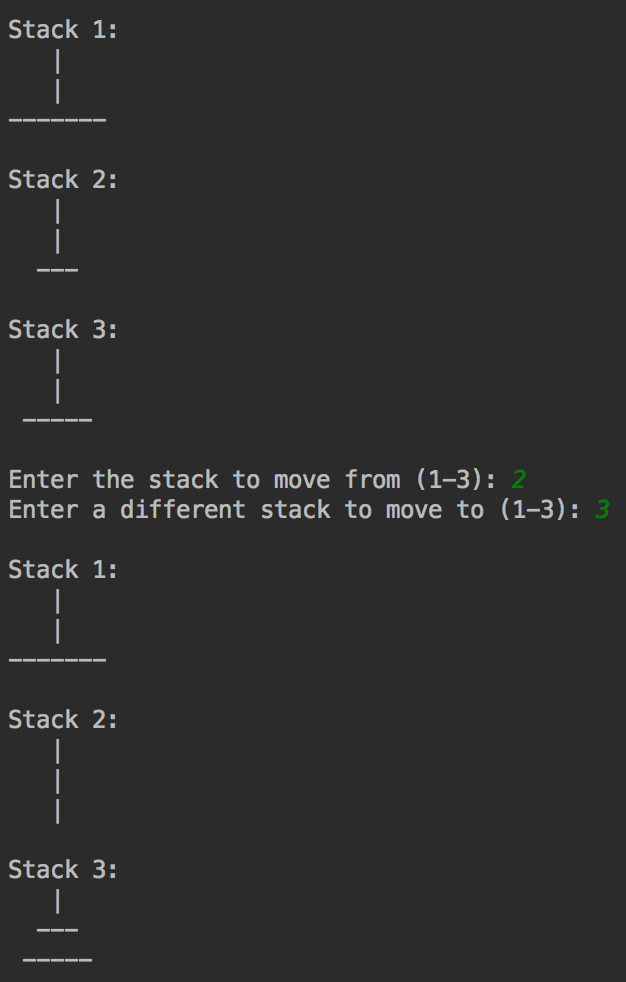
# Task 2: Single Player

The first three test cases are the following. Two other cases are included on the opposite page.

Clockwise from right:

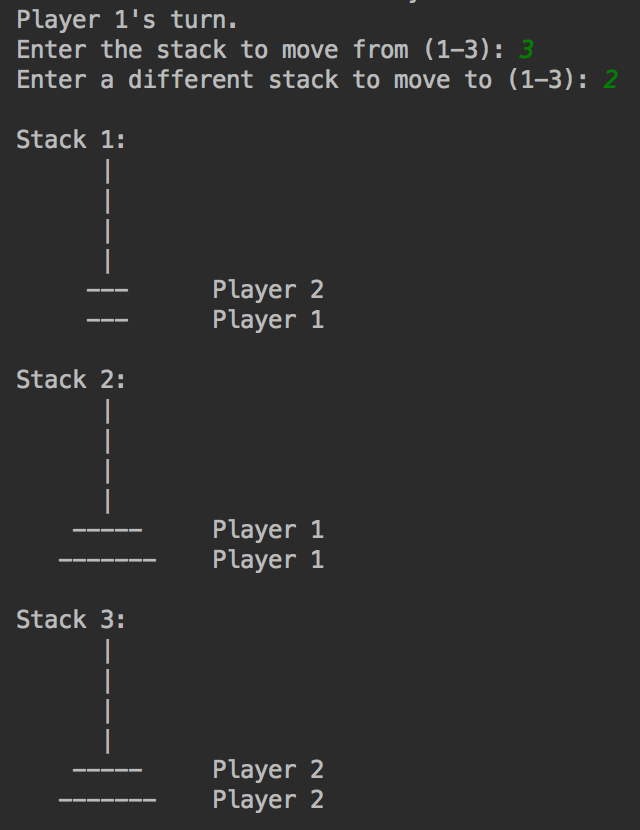
* Attempting to make an illegal move by placing a disk onto a smaller disk
* Making a legal move by placing a disk onto an empty stack
* Making an illegal move by attempting to move a disk from an empty stack

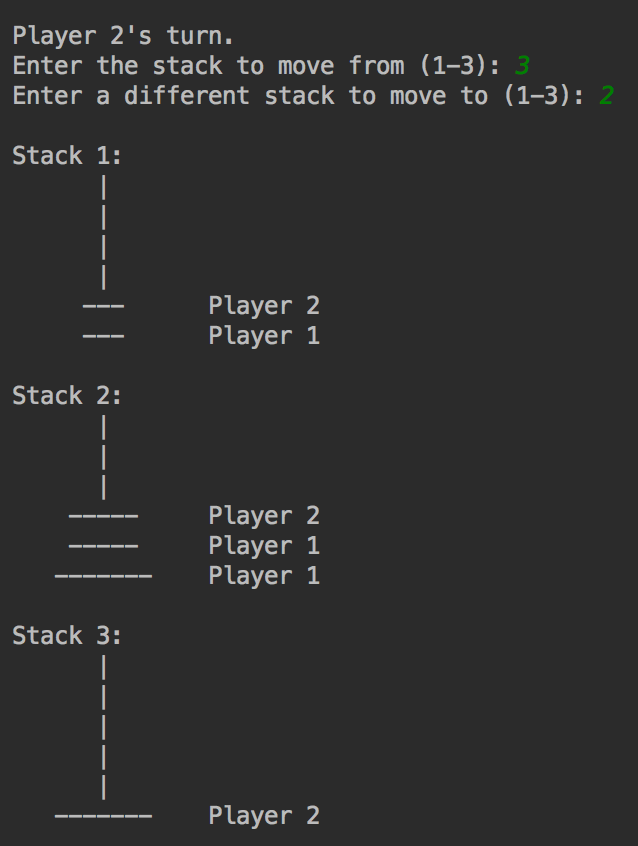


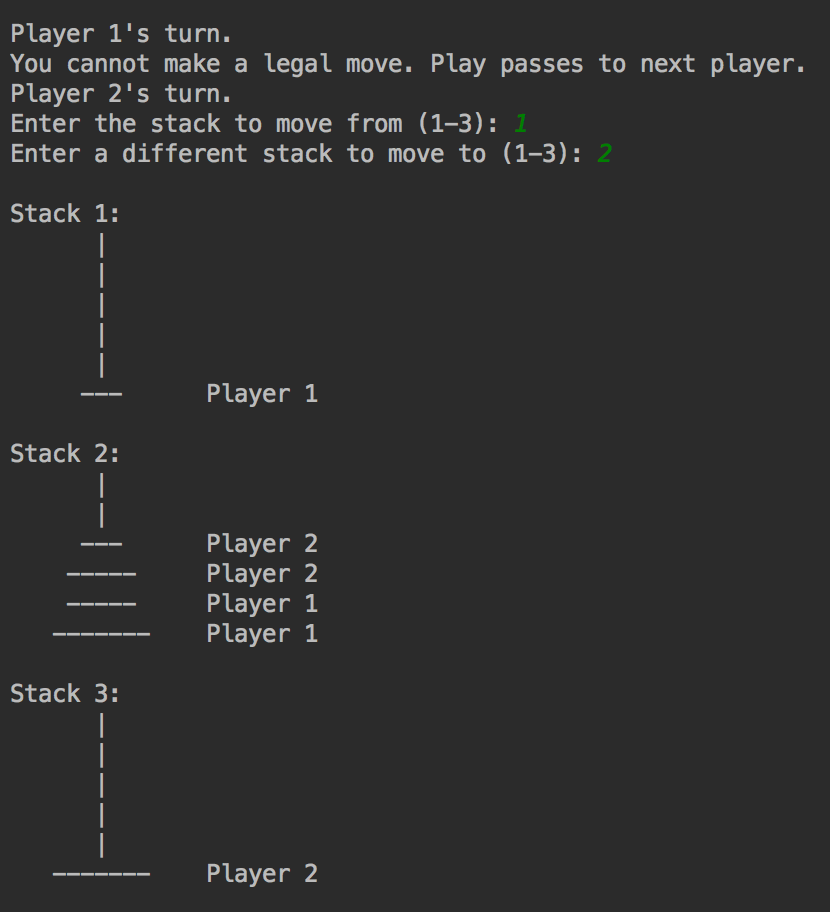


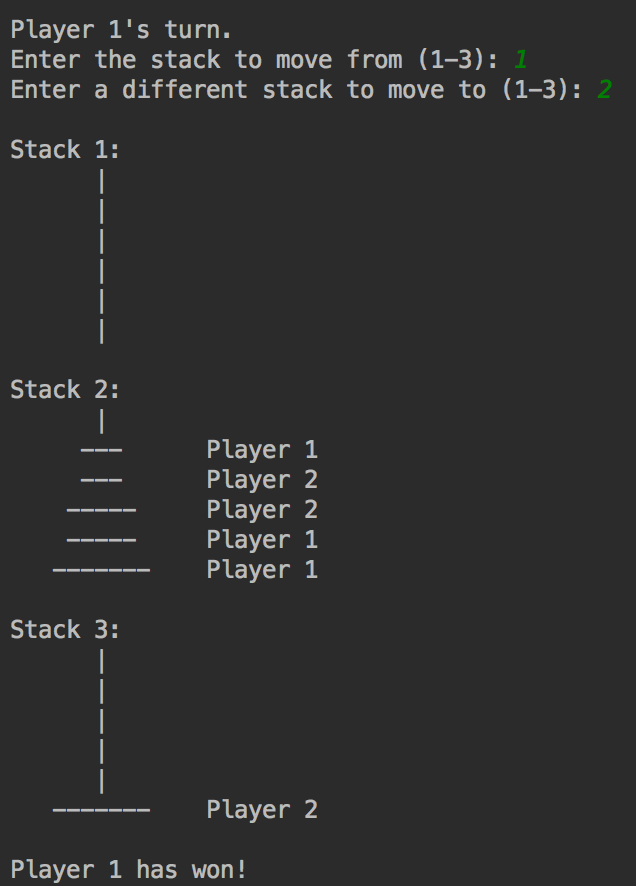
* Left: successfully moving a disk from one stack to another stack with a disk already on it
* Right: successfully winning the game

# Task 3: Two Players

Last 4 moves to win – two players, 3 disks each





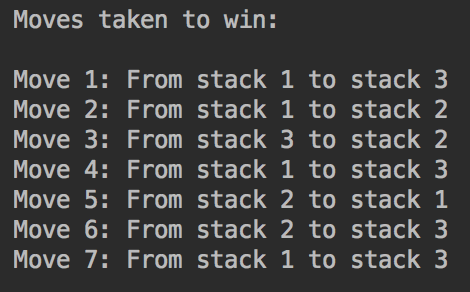


When attempting to play with two players, a collaborative playing style is almost necessary for either player to win. It is quite easy—even with only three disks per player—for a stalemate to result where one player can prevent any progress of the other and herself, either accidentally or deliberately. This can be accomplished by immediately placing, whenever possible, a disk over the other player’s largest disk. This prevents either end stack (1 or 3) from ever being cleared, which is required for a win.

It is possible that, if each player ignored the other and both went solely for the middle, that one player could get their largest plate to the other end from it being their only option. However, the best strategy to get a winner (for three disks) appears to be getting both players’ smallest disks on one stack and their other disks on separate stacks. This allows one player to win unavoidably, as the other is blocked from making any moves that would stop a win. See the above steps for the specific positioning and the moves involved.

# Task 4: Queue

Steps taken to win a game with three disks:



Using the STL vector provided a few challenges when implementing a queue. As you cannot pop items off the front of the queue, adding new items to the end would require the entire vector to be shifted left on every access. Therefore, I took advantage of the insert method of the class to enqueue items at the beginning. The items added first are then located the very end of the vector, where they can be accessed and popped off.

The other assorted functions (length, emptying the vector, checking if it is empty, peeking at the top item) are passed through to the vector, which has functions of its own to handle state and accessing it. I chose to use this over implementing their functionality again as the class most likely has the fastest possible code for each function.

As dequeuing and peeking are undefined if the queue is empty, I finally added an EmptyQueue exception to handle this case instead of forcing the user of the class to consider it.