**Classes**

**Node**

This class represents a group that has a name and a size (number of people in the group), which is extracted from the input text file. A pointer to a Node object is also added as a member variable to keep track of the order of input to a linked list (which is implemented in the queue and stack classes). The class has no member functions (besides the constructor and destructor), and everything is public.

Member Structure:

Public

* Name (string)
* Size (int)
* Next (Node\*)
* Node(string, int): constructor that sets the Name and Size member variables to parameters and next to nullptr

**Queue**

This class keeps track of Node objects (previously defined) based on input order (first in, first out). The only member variable in this class is a private pointer to a Node object, which represents the oldest Node object added to the linked list. The queue then has 6 public member functions, explained below:

Member Structure:

Private

* Top (Node\*)

Public

* Queue(): Creates a Queue object with the top variable set to nullptr
* ~Queue(): destroys any dynamic memory allocated during push operations if the queue is not empty when the program finished running
* Node\* peek(): returns a pointer to the top (oldest) value in the queue
* Void dequeue(): deallocates memory from node inserted in to the queue and reassigns the top pointer
* Void enqueue(string, int): dynamically allocates memory to create a Node object, and it changes the “Next” pointer of the last object in the queue from null to the newly created Node
* int length(): returns an integer representing the number of Nodes in the queue

**Stack**

This class keeps track of Node objects (previously defined) based on input order (first in, last out). The only member variable in this class is a private pointer to a Node object, which represents the oldest Node object added to the linked list. The queue then has 6 public member functions, explained below:

Member Structure:

Private

* Top (Node\*)

Public

* Stack(): Creates a Stack object with the top variable set to nullptr
* ~Stack (): destroys any dynamic memory allocated during push operations if the queue is not empty when the program finished running
* Node\* peek(): returns a pointer to the top (oldest) value in the queue
* Void pop(): deallocates memory from node inserted in to the stack. “Top” pointer is assigned to the value that came in the latest.
* Void push(string, int): dynamically allocates memory to create a node object, and it assigns the “top” pointer to this new node. The “Next” pointer of the new node is assigned to the value previously as the top of the stack
* int length(): returns an integer representing the number of Nodes in the stack

**Main File Implementation**

The approach I took to using the queue and stack to produce a file that has groups in reversed order was the following:

1. Create variables that keeps track of the current capsule being filled, and the total sum of people that the current capsule has.
2. Read the input file and store all the groups as nodes in the queue.
3. Start a while loop that ends once the queue is empty.
4. In the loop, peek into the queue to get the top value. Check if the total sum of people + the node.size will be greater than 15. If smaller or equal than 15, dequeue the node and place it in the stack. If larger than 15, don’t dequeue it and call the writeOutput() function, which writes the current capsule stats to a text file.
5. Repeat this process until the queue is empty. There is a condition in the while loop that checks if the stack has values and the next node is null, which allows the stack to be emptied out if the queue is done.
6. Print the total stats of the scheduling using the endStats() method.

**Main Functions**:

writeOutput(Stack&, int capCount, int capNum): This function is called when a capsule is full and we need to write to a file. It empties out the stack as well. It takes a reference to the stack, the capCount which is the capsule count for the number of people, and the capsule number.

endStats(float totalPpl, float totalCap): prints the final stats, such as the number of guests, capsules, and the percentage utilization. The totalPpl float variable has the total number of guests, and the totalCap has the total number of capsules. This is used to calculate percentage utilization.

**Improvements**

The “next” variable of Node could be made private to prevent misuse of the pointer in the user end. This would require creating an accessor and mutator for it. However, the implementation is simple enough that making extra functions would complicate the implementation.

A valid improvement would be to return a copy of a Node variable rather than a pointer to an object in the peek() function. I encountered an issue with memory management while debugging because the value in the memory address is deleted after calling dequeue, so the pointer to the Node becomes useless. Returning a copy of the top Node variable would avoid this issue, and we could even set “next” to nullptr, further preventing memory misuse. This would be acceptable because all we are interested in is the group name and size; this also addresses the previous issue of pointer misuse of “next”; we could make the “next” variable private and avoid the issue.

**Instructions for Compiling:**

Get all the code submitted (DarkZone.cpp, Stack.cpp, Stack.h, Queue.cpp, Queue.h, DarkZone.txt, makefile) in one folder. In your terminal, go to the folder where everything is saved (ex: cd “Project 3”), and then type “make” in your terminal. This will create a link all the files and create a.out, which is the executable file. Then, run “./a.out” in the terminal. This should execute the linked file and run it.