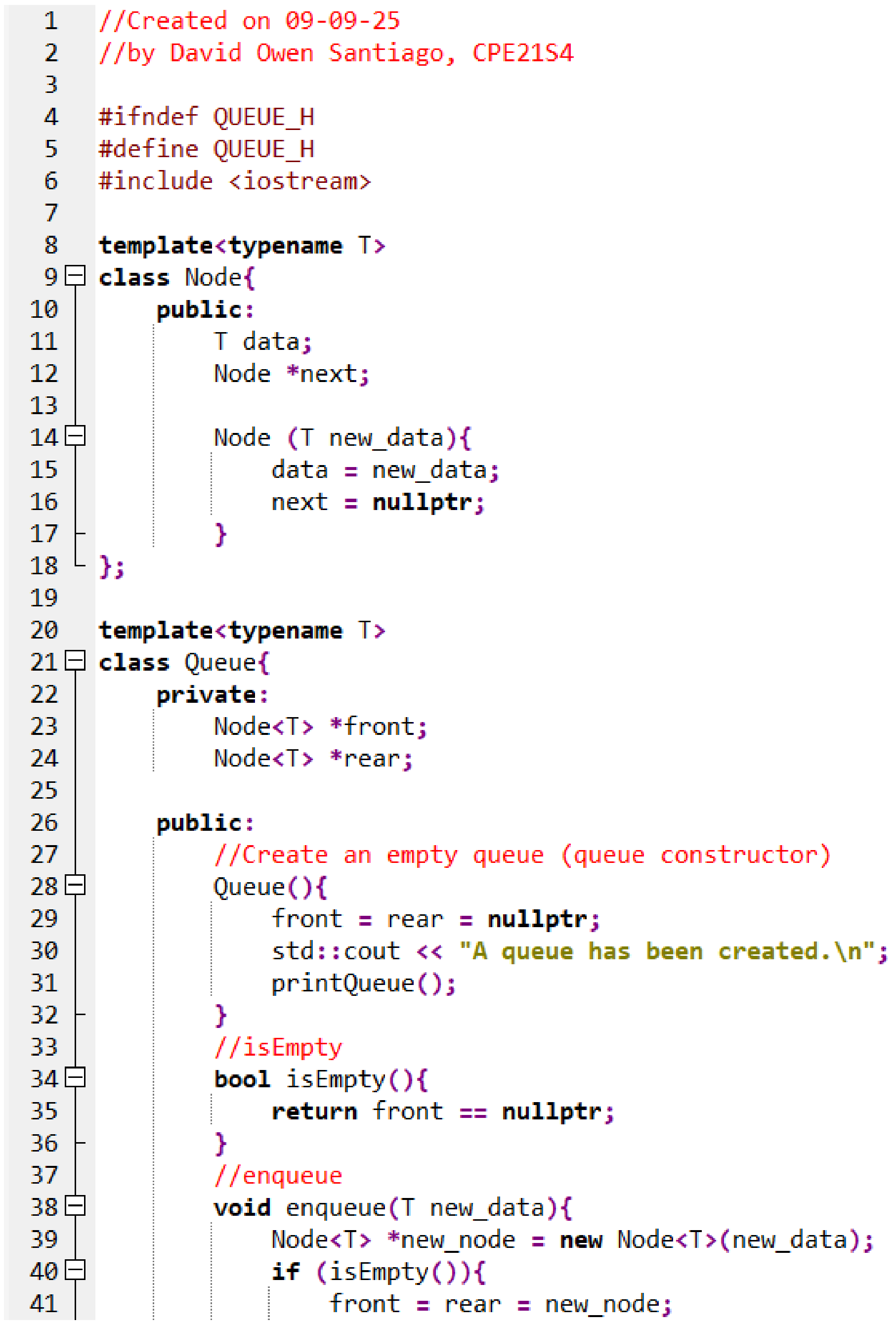
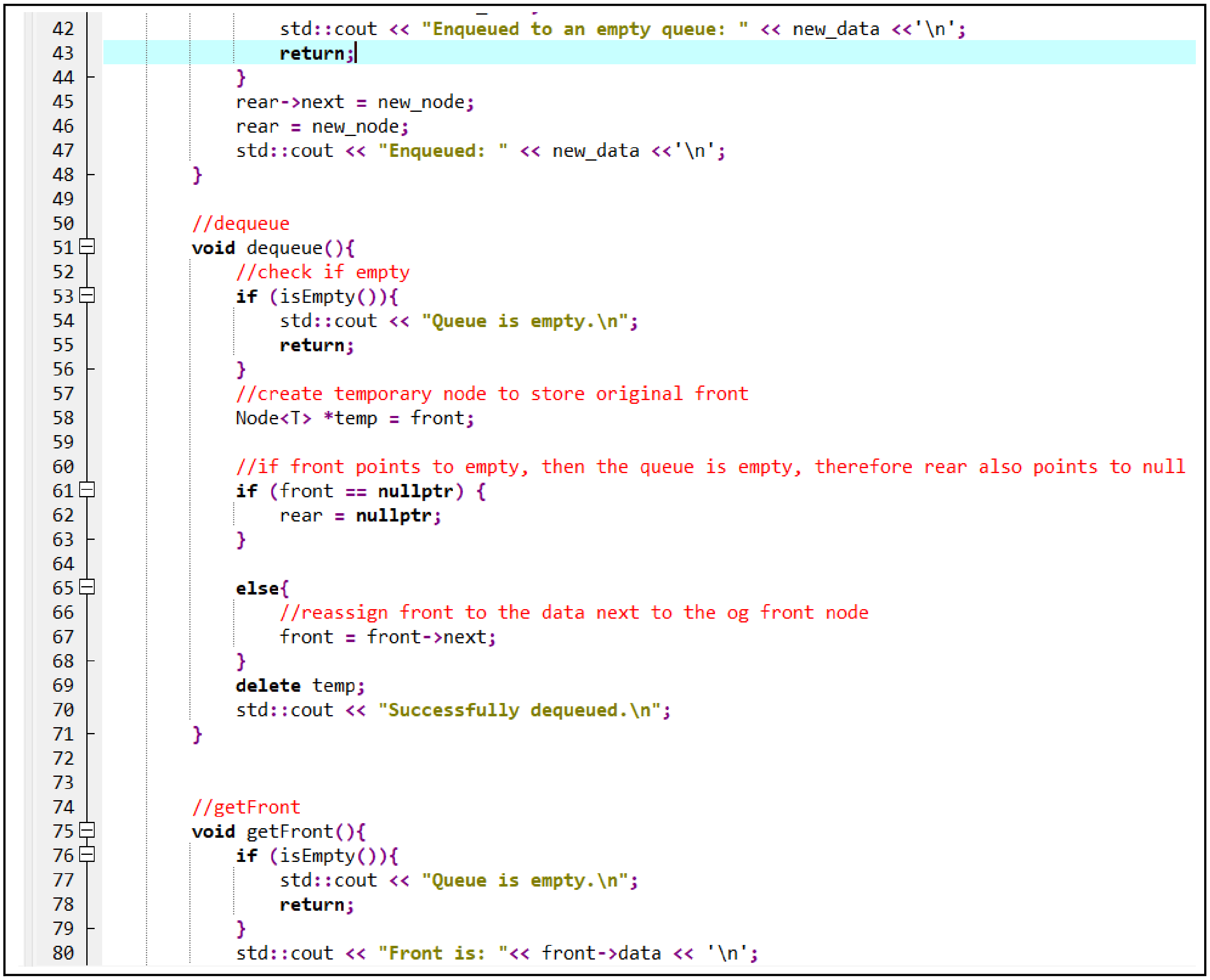
Seatwork 5.1	
Queue - Linked List Application	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09 - 09 - 25
Section: CPE21S4	**Date Submitted:** 09 - 09 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado
6. Output	

Source Code:

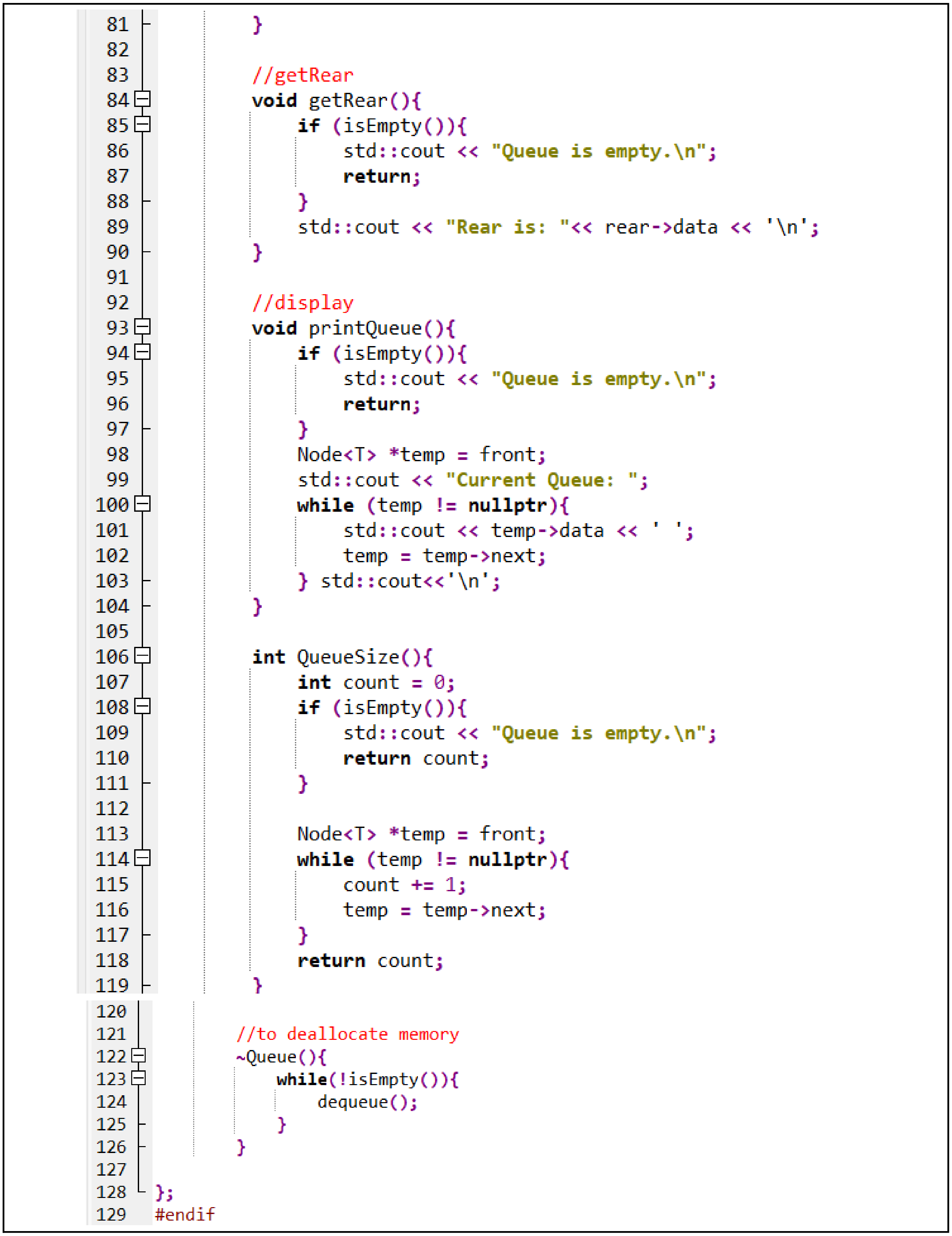
Header File:



```
//Created on 09-09-25
//by David Owen Santiago, CPE21S4

#ifndef QUEUE_H
#define QUEUE_H
#include <iostream>

template<typename T>
class Node{
    public:
        T data;
        Node *next;

        Node (T new_data){
            data = new_data;
            next = nullptr;
        }
};

template<typename T>
class Queue{
    private:
        Node<T> *front;
        Node<T> *rear;

    public:
        //Create an empty queue (queue constructor)
        Queue(){
            front = rear = nullptr;
            std::cout << "A queue has been created.\n";
            printQueue();
        }
        //isEmpty
        bool isEmpty(){
            return front == nullptr;
        }
        //enqueue
        void enqueue(T new_data){
            Node<T> *new_node = new Node<T>(new_data);
            if (isEmpty()){
                front = rear = new_node;
```

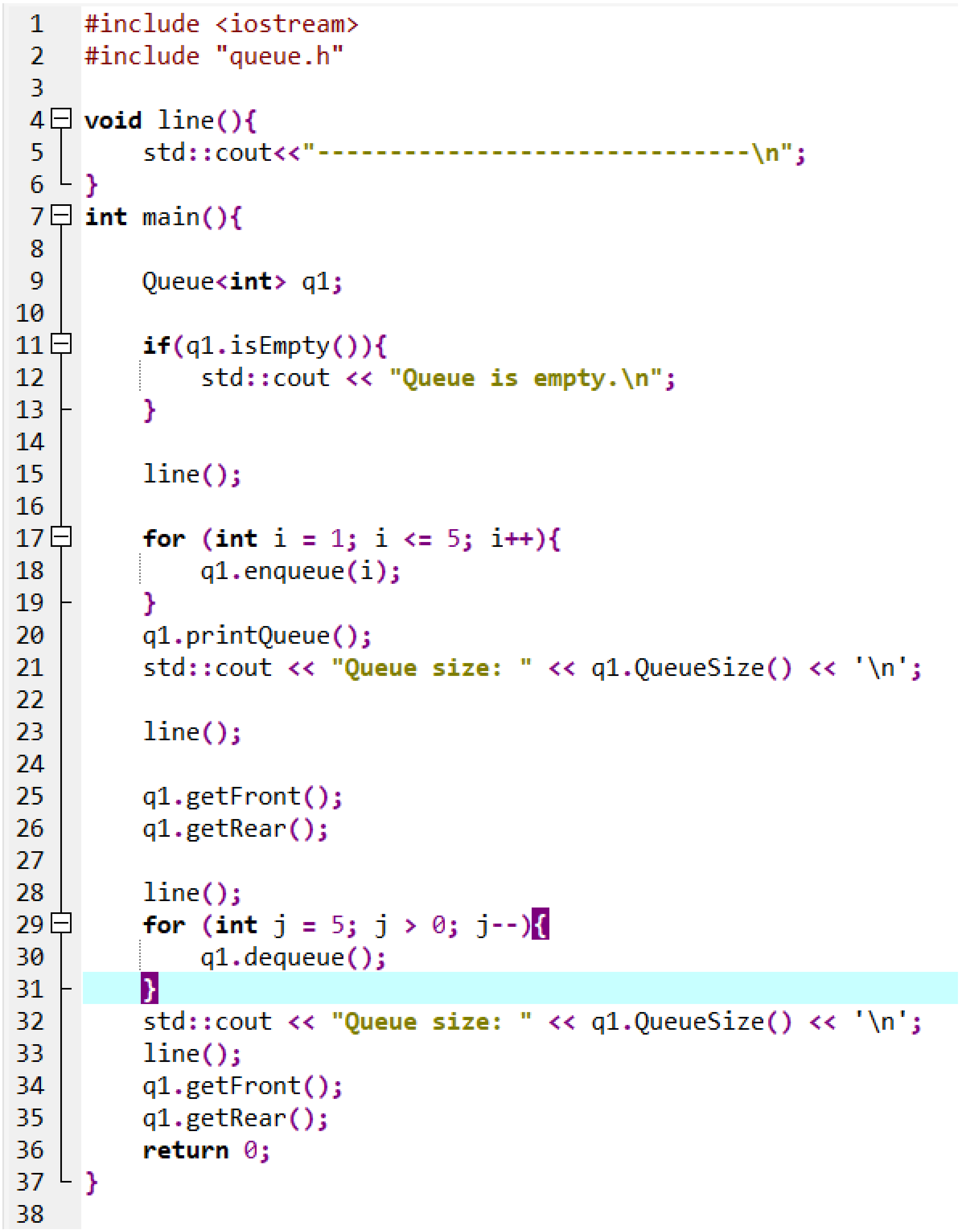


```cpp
                std::cout << "Enqueued to an empty queue: " << new_data <<'\n';
                return;
            }
            rear->next = new_node;
            rear = new_node;
            std::cout << "Enqueued: " << new_data <<'\n';
        }

        //dequeue
        void dequeue(){
            //check if empty
            if (isEmpty()){
                std::cout << "Queue is empty.\n";
                return;
            }
            //create temporary node to store original front
            Node<T> *temp = front;

            //if front points to empty, then the queue is empty, therefore rear also points to null
            if (front == nullptr) {
                rear = nullptr;
            }

            else{
                //reassign front to the data next to the og front node
                front = front->next;
            }
            delete temp;
            std::cout << "Successfully dequeued.\n";
        }


        //getFront
        void getFront(){
            if (isEmpty()){
                std::cout << "Queue is empty.\n";
                return;
            }
            std::cout << "Front is: "<< front->data << '\n';
```



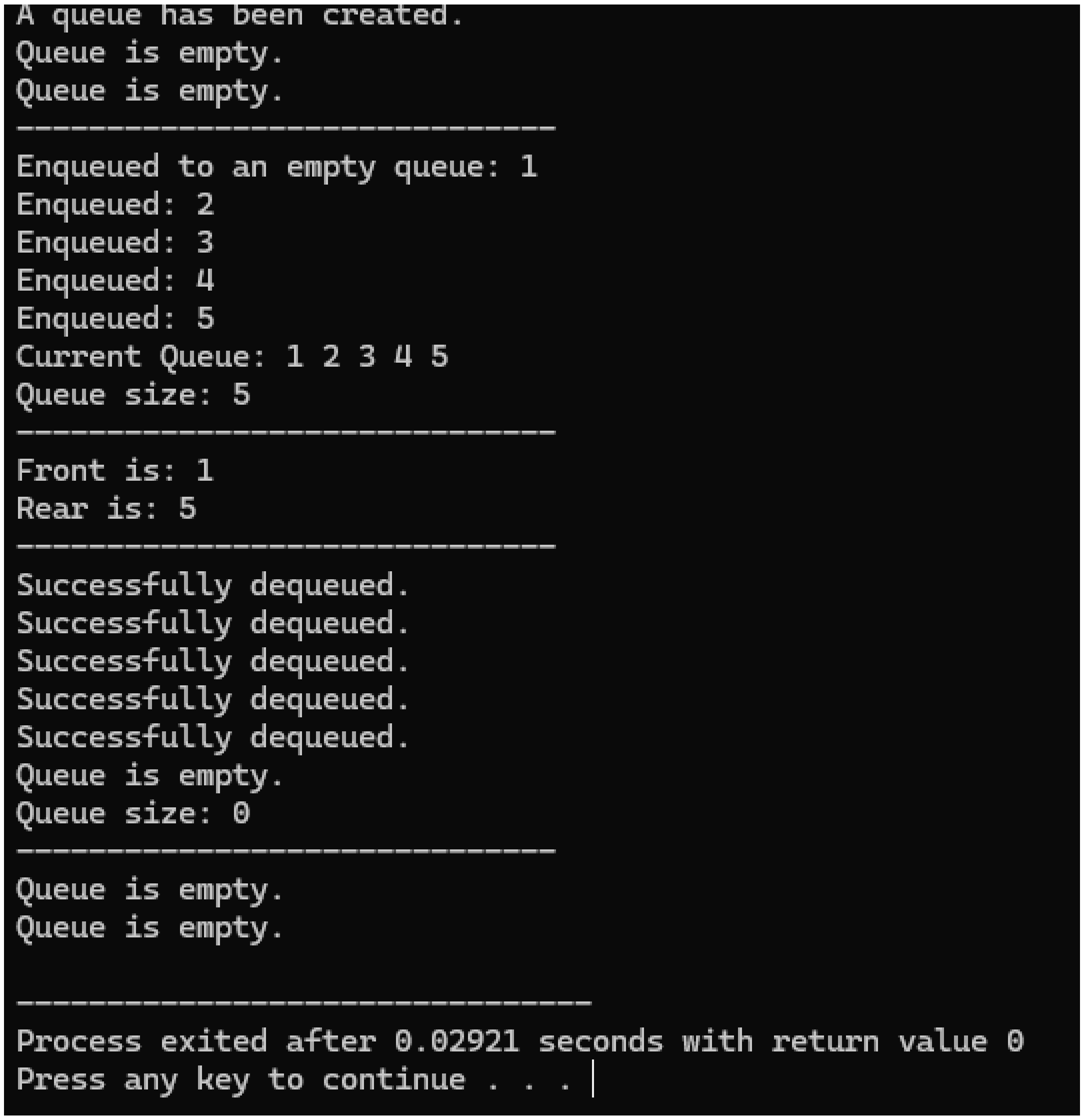
```cpp
    }

    //getRear
    void getRear(){
        if (isEmpty()){
            std::cout << "Queue is empty.\n";
            return;
        }
        std::cout << "Rear is: "<< rear->data << '\n';
    }

    //display
    void printQueue(){
        if (isEmpty()){
            std::cout << "Queue is empty.\n";
            return;
        }
        Node<T> *temp = front;
        std::cout << "Current Queue: ";
        while (temp != nullptr){
            std::cout << temp->data << ' ';
            temp = temp->next;
        } std::cout<<'\n';
    }

    int QueueSize(){
        int count = 0;
        if (isEmpty()){
            std::cout << "Queue is empty.\n";
            return count;
        }

        Node<T> *temp = front;
        while (temp != nullptr){
            count += 1;
            temp = temp->next;
        }
        return count;
    }

    //to deallocate memory
    ~Queue(){
        while(!isEmpty()){
            dequeue();
        }
    }

};
#endif
```

Main file:



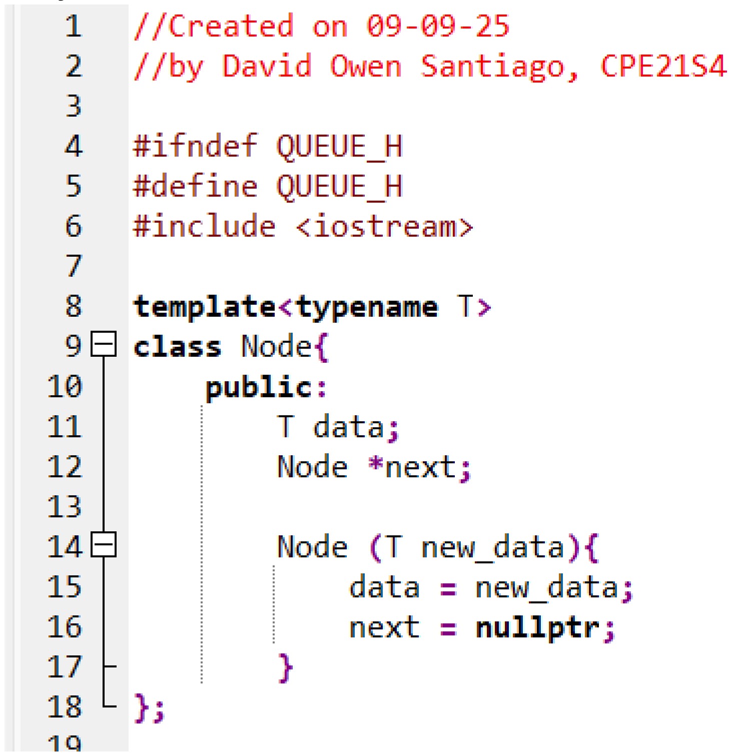
```cpp
#include <iostream>
#include "queue.h"

void line(){
    std::cout<<"-----------------------------\n";
}
int main(){

    Queue<int> q1;

    if(q1.isEmpty()){
        std::cout << "Queue is empty.\n";
    }

    line();

    for (int i = 1; i <= 5; i++){
        q1.enqueue(i);
    }
    q1.printQueue();
    std::cout << "Queue size: " << q1.QueueSize() << '\n';

    line();

    q1.getFront();
    q1.getRear();

    line();
    for (int j = 5; j > 0; j--){
        q1.dequeue();
    }
    std::cout << "Queue size: " << q1.QueueSize() << '\n';
    line();
    q1.getFront();
    q1.getRear();
    return 0;
}

```

Output:



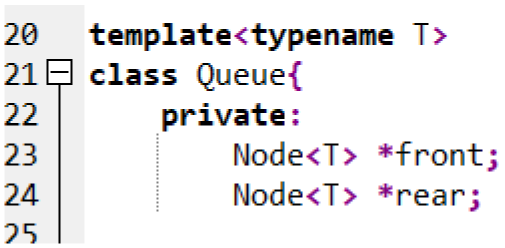
```
A queue has been created.
Queue is empty.
Queue is empty.
------------------------------
Enqueued to an empty queue: 1
Enqueued: 2
Enqueued: 3
Enqueued: 4
Enqueued: 5
Current Queue: 1 2 3 4 5
Queue size: 5
------------------------------
Front is: 1
Rear is: 5
------------------------------
Successfully dequeued.
Successfully dequeued.
Successfully dequeued.
Successfully dequeued.
Successfully dequeued.
Queue is empty.
Queue size: 0
------------------------------
Queue is empty.
Queue is empty.

--------------------------------
Process exited after 0.02921 seconds with return value 0
Press any key to continue . . .
```

Analysis:



```cpp
//Created on 09-09-25
//by David Owen Santiago, CPE21S4

#ifndef QUEUE_H
#define QUEUE_H
#include <iostream>

template<typename T>
class Node{
    public:
        T data;
        Node *next;

        Node (T new_data){
            data = new_data;
            next = nullptr;
        }
};
```

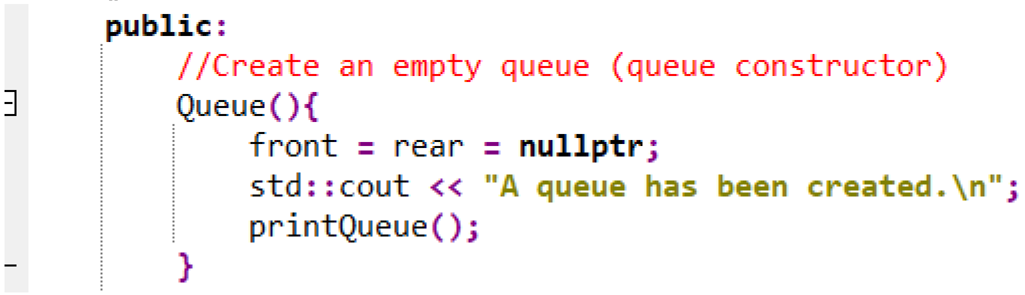
This section initializes the header file creation. We first create a class Node to prepare the linked list creation. In this we create an object data of type T which serves as the data of each node. Afterwards, we create a new Node pointer that will point to next. After those two are created, we then make a Node constructor that takes a type T data as its parameter, which will become the new data of the new node. Since this is a queue, the node next to the new node will always point to null.



```cpp
template<typename T>
class Queue{
    private:
        Node<T> *front;
        Node<T> *rear;

```

This is the initialization of the class Queue with the private members being front and rear since all of the operations revolve around these two pointers. They are private so that they cannot be directly accessed or changed outside of the class.

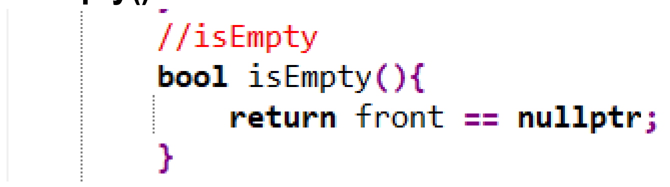
Queue()



```cpp
    public:
        //Create an empty queue (queue constructor)
        Queue(){
            front = rear = nullptr;
            std::cout << "A queue has been created.\n";
            printQueue();
        }
```

This is the queue constructor—it initializes the values of the front and rear. Since it begins as an empty queue, they must all point to null.

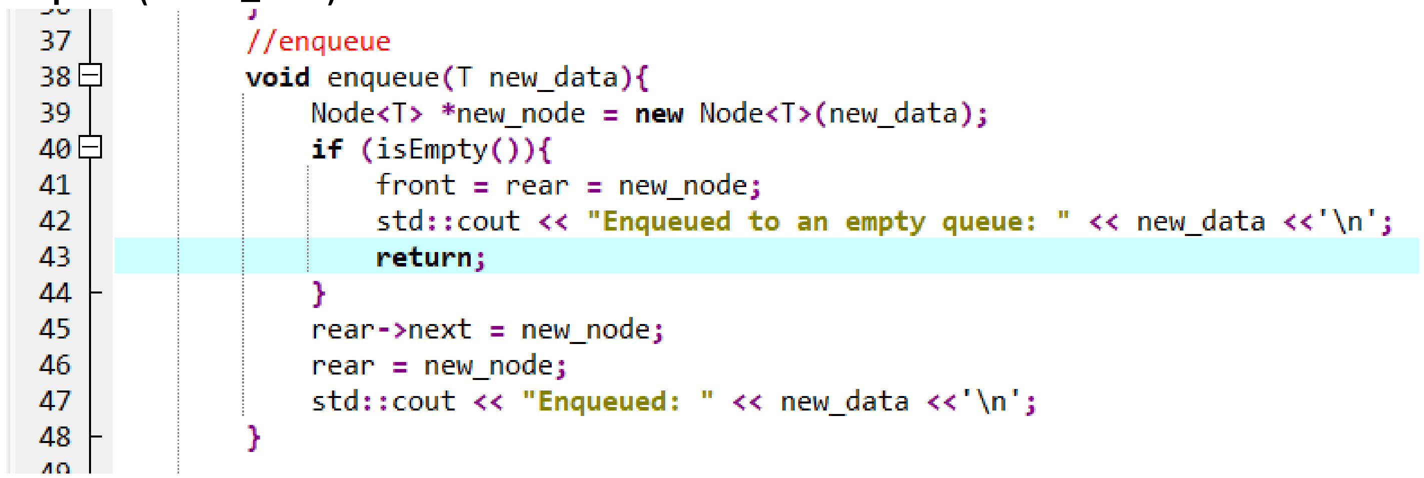
isEmpty()



```
//isEmpty
bool isEmpty(){
    return front == nullptr;
}
```

This function will return true if front points to null, which means that it is empty.

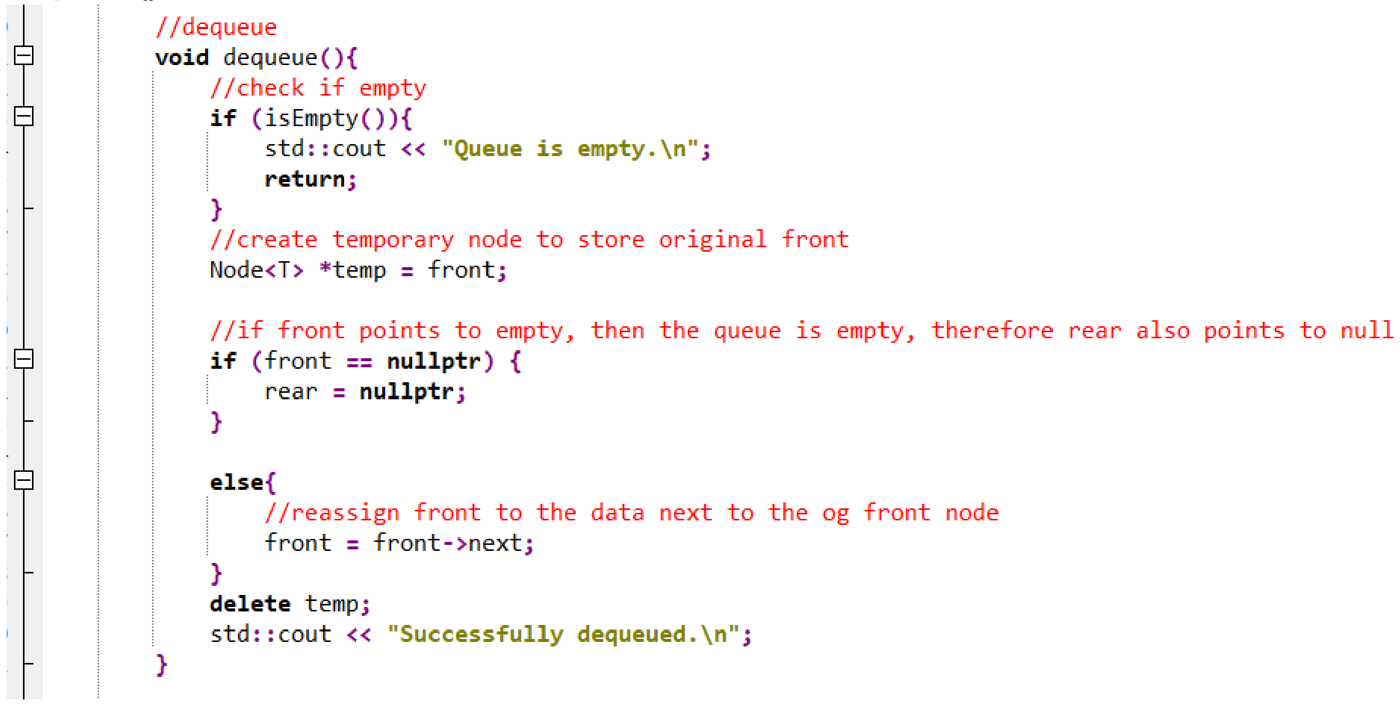
enqueue(T new_data)



```
//enqueue
void enqueue(T new_data){
    Node<T> *new_node = new Node<T>(new_data);
    if (isEmpty()){
        front = rear = new_node;
        std::cout << "Enqueued to an empty queue: " << new_data <<'\n';
        return;
    }
    rear->next = new_node;
    rear = new_node;
    std::cout << "Enqueued: " << new_data <<'\n';
}
```

This is the enqueue function—it creates and enters a new node at the rear and it becomes the new rear. It begins by declaring a new node pointer with its type as T (since it is a header file) that equals a new Node. We call the Node constructor to create a new node with its data being the new_data. Afterwards, an if statement must be fulfilled. If the queue is empty, then the front and rear will become the new data, then the function ends. Otherwise, meaning the queue is not empty, the rear->next value will be the new node then the new_node will become the new rear.

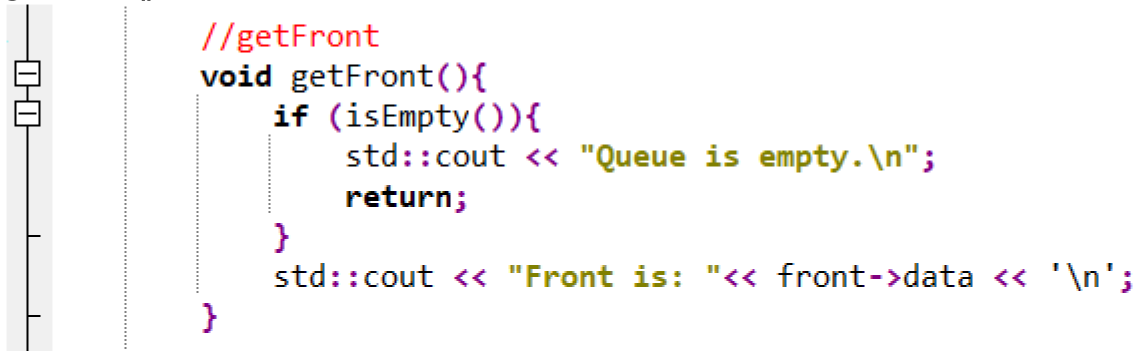
dequeue()



```
//dequeue
void dequeue(){
    //check if empty
    if (isEmpty()){
        std::cout << "Queue is empty.\n";
        return;
    }
    //create temporary node to store original front
    Node<T> *temp = front;

    //if front points to empty, then the queue is empty, therefore rear also points to null
    if (front == nullptr) {
        rear = nullptr;
    }

    else{
        //reassign front to the data next to the og front node
        front = front->next;
    }
    delete temp;
    std::cout << "Successfully dequeued.\n";
}
```

This function deletes the node at the front. It begins by checking if the queue is empty and prints a corresponding message. Otherwise, we create a temporary node to store the original front which will be deleted later. An if-else statement first checks if the front points to null, which if it does, then the rear will also point to null. Else, the front will then change to the data next to it. Afterwards, we deallocate the memory of the temp variable which stores the original front to using the delete keyword.

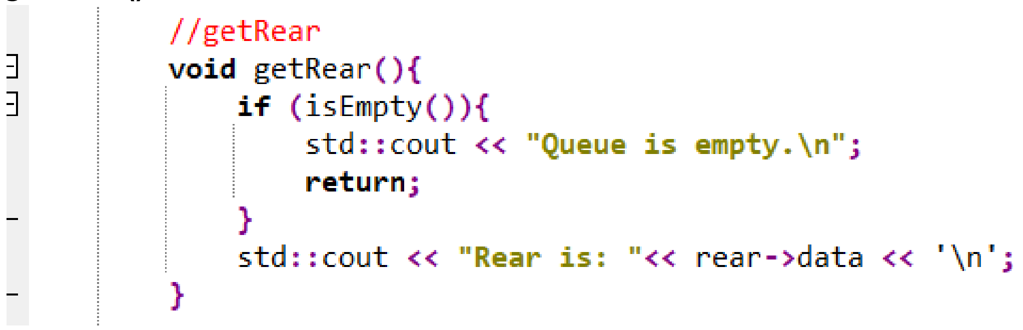
getFront()



```cpp
//getFront
void getFront(){
    if (isEmpty()){
        std::cout << "Queue is empty.\n";
        return;
    }
    std::cout << "Front is: "<< front->data << '\n';
}
```

This is similar to the peek() function in stack, where we only view the value of the front node. If it isn't empty, then we simply print the data stored in the address of the front.

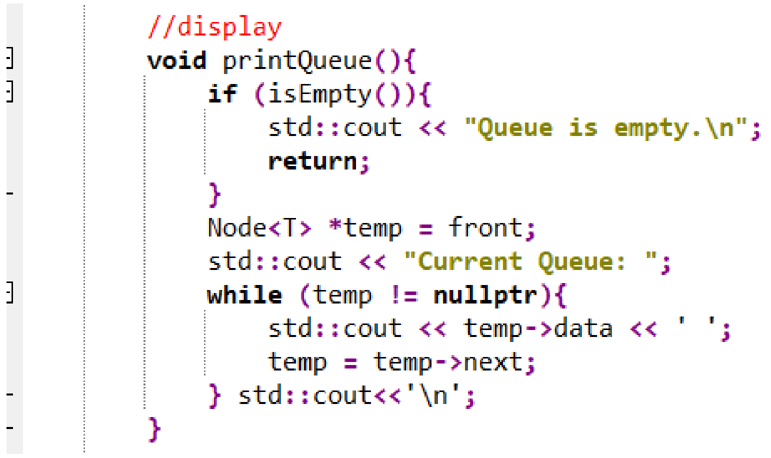
getRear()



```cpp
//getRear
void getRear(){
    if (isEmpty()){
        std::cout << "Queue is empty.\n";
        return;
    }
    std::cout << "Rear is: "<< rear->data << '\n';
}
```

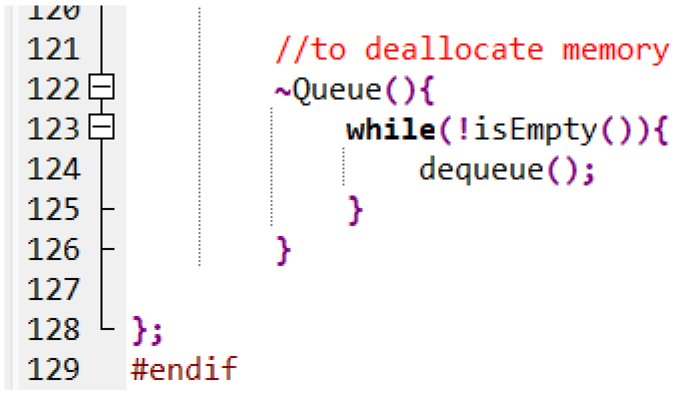
This works exactly the same as the getFront function, except it prints out the rear or last node's data instead.

printQueue()



```cpp
//display
void printQueue(){
    if (isEmpty()){
        std::cout << "Queue is empty.\n";
        return;
    }
    Node<T> *temp = front;
    std::cout << "Current Queue: ";
    while (temp != nullptr){
        std::cout << temp->data << ' ';
        temp = temp->next;
    } std::cout<<'\n';
}
```

This function, if not empty, first creates a temporary node that stores the front. Since we're only printing, we don't want to modify the actual queue, and in order to do that, we use a temporary variable. Once created, we initialize a while loop that runs true while temp doesn't point to null. With each iteration of the loop, the data is printed then the temp is updated to store the next value, again, until it points to null. This should print out all of the data in the current queue.



```
        //to deallocate memory
        ~Queue(){
            while(!isEmpty()){
                dequeue();
            }
        }

};
#endif
```

This is a deconstructor that clears all of the memory currently used by the program after it ends. It uses a while loop that keeps dequeueing, deallocating all of the memory that the queue occupies.

7. Supplementary Activity

8. Conclusion

In conclusion, this activity serves as a proper hands-on creation of the linked-list implementation of a queue. I learned the actual logic per line of code which helped me understand how it works and how each function is implemented. I was also able to connect the previous stack and linked list lessons in this activity which made it easier to understand. Moreover, I now understand how to properly implement a constructor and deconstructor in a header file and class. Additionally, since it is created using a header file, I was also able to make it modular so that I can simply load the header file in a main function.

9. Assessment Rubric