Activity No. 5	
QUEUES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: OCTOBER 7, 2024**
Section: CPE21S1	**Date Submitted: OCTOBER , 2024**
Name(s): GASPAR, AARON ROWEN O.	**Instructor: MS. MARIA RIZETTE SAYO**
6. Output	

Table 5-1. Queues using C++ STL

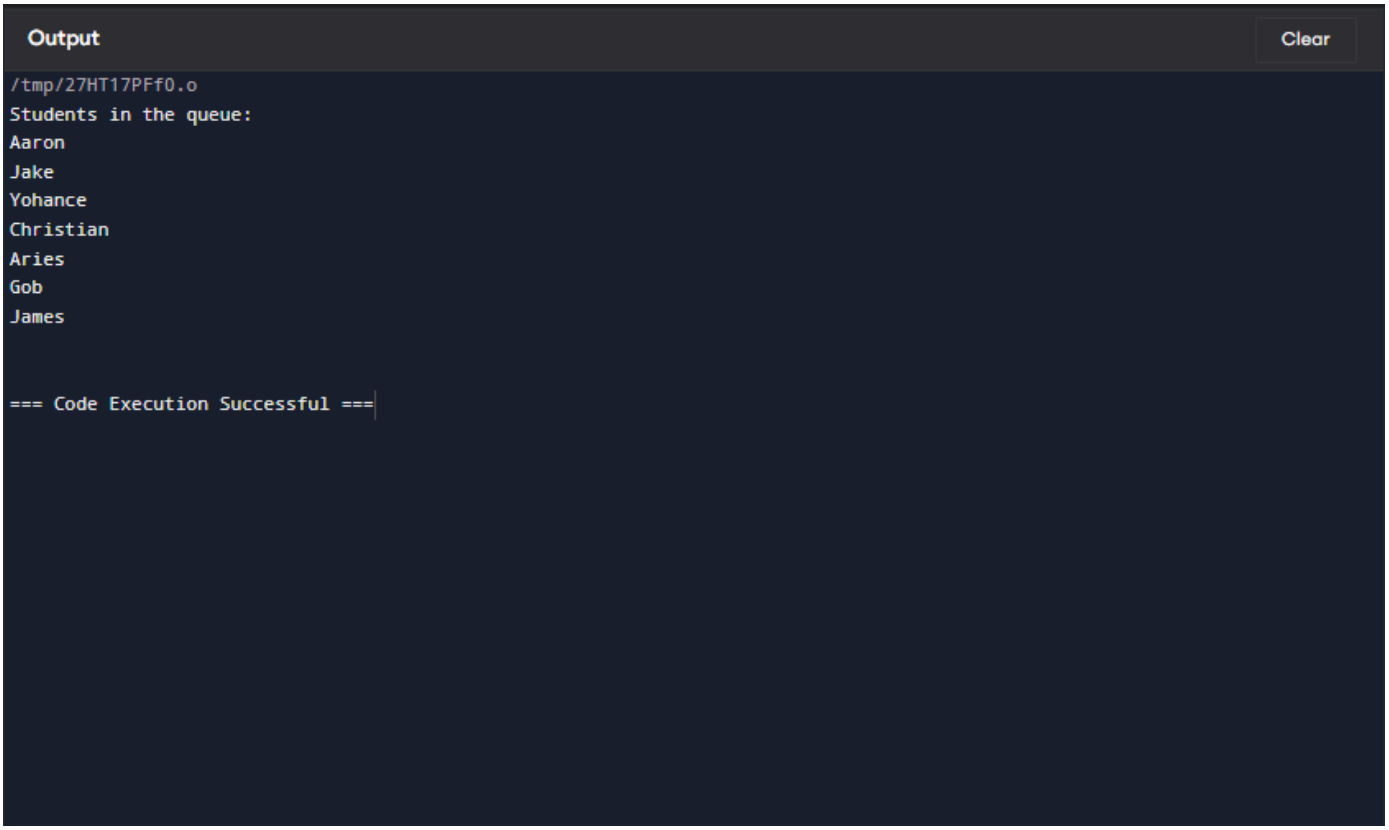


Table 5-2. Queues using Linked List Implementation

main.cpp

Share Run

```cpp
#include <iostream>
#include <string>
using namespace std;

struct Node {
    string data;
    Node* next;
};


class Queue {
private:
    Node* front;
    Node* rear;
    int size;

public:

    Queue() {
        front = rear = nullptr;
        size = 0;
    }


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


    int getSize() {
        return size;
    }
```

```
35      void enqueue(string student) {
36          Node* temp = new Node();
37          temp->data = student;
38          temp->next = nullptr;
39
40          if (isEmpty()) {
41              front = rear = temp;
42          } else {
43              rear->next = temp;
44              rear = temp;
45          }
46
47          size++;
48          cout << "Inserted on the QUEUE: " << student << "\n";
49      }
50
51
52      void dequeue() {
53          if (isEmpty()) {
54              cout << "No one on the QUEUE. \n";
55              return;
56          }
57
58          Node* temp = front;
59          front = front->next;
60
61          if (front == nullptr)
62              rear = nullptr;
63
64          cout << "Deleted on the QUEUE: " << temp->data << "\n";
65          delete temp;
66
67          size--;
68      }
```

```
    string getFront() {
        if (!isEmpty())
            return front->data;
        return "Queue is empty";
    }
};

int main() {
    Queue studentsQueue;

    cout << "Inserting into an empty queue\n";
    studentsQueue.enqueue("Aaron");
    cout << "size: " << studentsQueue.getSize() << "\n\n";

    cout << "Inserting into a non-empty queue\n";
    studentsQueue.enqueue("Jake");
    cout << "size: " << studentsQueue.getSize() << "\n\n";

    cout << "Deleting from a queue with more than one item\n";
    cout << "Front of the queue before deletion: " << studentsQueue.getFront() << "\n";
    studentsQueue.dequeue();
    cout << "size: " << studentsQueue.getSize() << "\n";
    cout << "Front of the queue after deletion: " << studentsQueue.getFront() << "\n\n";

    cout << "Deleting from a queue with one item\n";
    cout << "Front of the queue before deletion: " << studentsQueue.getFront() << "\n";
    studentsQueue.dequeue();
    cout << "Queue is now empty: " << (studentsQueue.isEmpty() ? "True" : "False") << "\n";

    return 0;
}
```

Output Clear

```
/tmp/NVssl6sWrp.o
Inserting into an empty queue
Inserted: Aaron
Queue size: 1

Inserting into a non-empty queue
Inserted: Jake
Queue size: 2

Deleting from a queue with more than one item
Front of the queue before deletion: Aaron
Deleted: Aaron
Queue size: 1
Front of the queue after deletion: Jake

Deleting from a queue with one item
Front of the queue before deletion: Jake
Deleted: Jake
Queue is now empty: True


=== Code Execution Successful ===
```

Table 5-3. Queues using Array Implementation

main.cpp

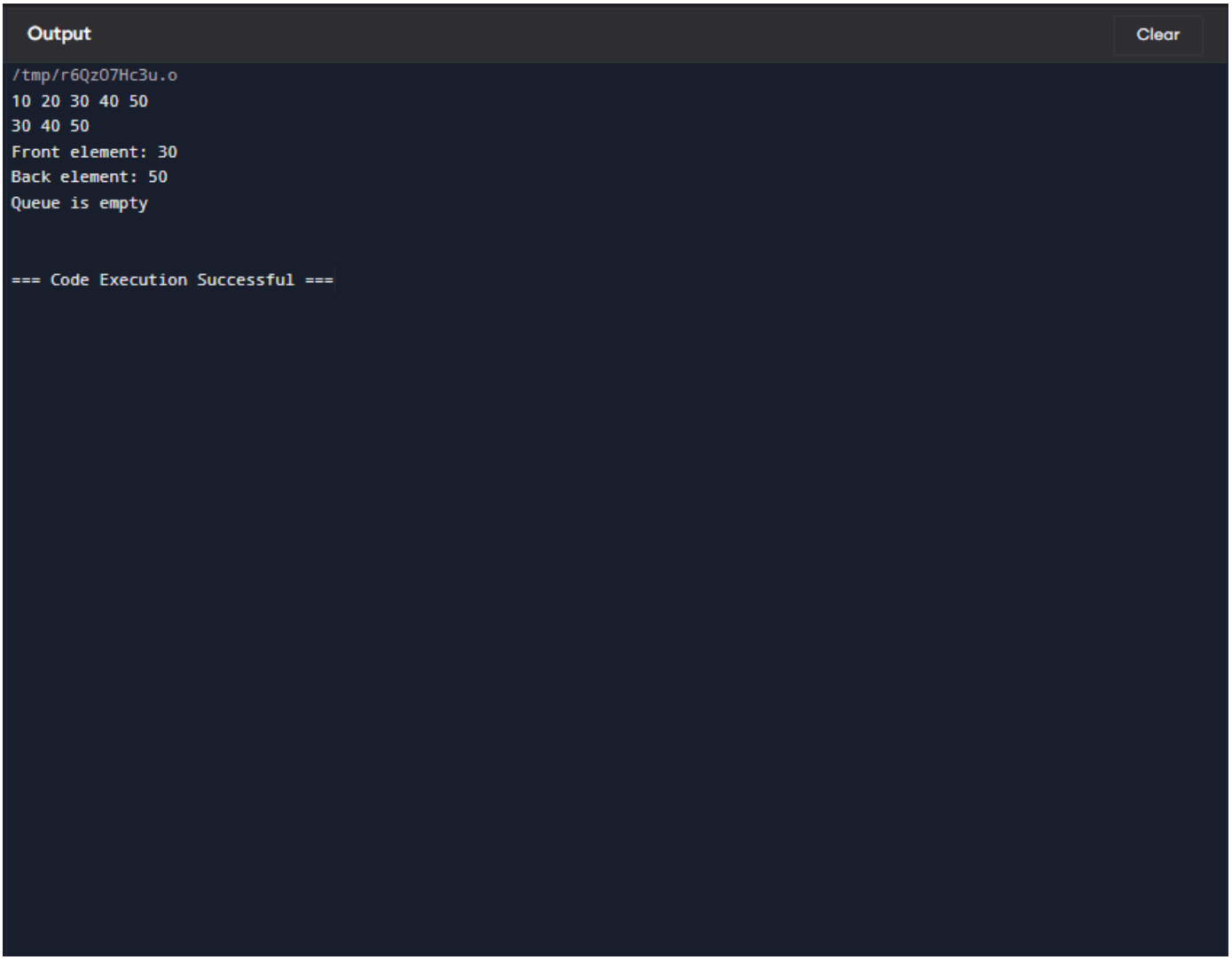
Share Run

```cpp
#include <iostream>
using namespace std;

class Queue {
private:
    int* queue;
    int capacity;
    int front;
    int rear;
    int size;

public:
    // Constructor
    Queue(int c) : capacity(c), front(0), rear(0), size(0) {
        queue = new int[capacity];
    }

    // Copy Constructor
    Queue(const Queue& other) : capacity(other.capacity), front(other.front), rear(other.rear), size(other.size) {
        queue = new int[capacity];
        for (int i = 0; i < capacity; ++i) {
            queue[i] = other.queue[i];
        }
    }

    // Destructor
    ~Queue() {
        delete[] queue;
    }
```

```
    // Copy Assignment Operator
    Queue& operator=(const Queue& other) {
        if (this != &other) {
            delete[] queue;
            capacity = other.capacity;
            front = other.front;
            rear = other.rear;
            size = other.size;
            queue = new int[capacity];
            for (int i = 0; i < capacity; ++i) {
                queue[i] = other.queue[i];
            }
        }
        return *this;
    }

    // Check if the queue is empty
    bool isEmpty() const {
        return size == 0;
    }

    // Return the size of the queue
    int getSize() const {
        return size;
    }

    // Clear the queue
    void clear() {
        front = 0;
        rear = 0;
        size = 0;
    }
```

```
    // Access the front element
    int frontElement() const {
        if (isEmpty()) {
            throw out_of_range("Queue is empty");
        }
        return queue[front];
    }

    // Access the back element
    int backElement() const {
        if (isEmpty()) {
            throw out_of_range("Queue is empty");
        }
        return queue[(rear - 1 + capacity) % capacity];
    }

    // Enqueue an element
    void enqueue(int data) {
        if (size == capacity) {
            throw overflow_error("Queue is full");
        }
        queue[rear] = data;
        rear = (rear + 1) % capacity;
        size++;
    }

    // Dequeue an element
    void dequeue() {
        if (isEmpty()) {
            throw underflow_error("Queue is empty");
        }
        front = (front + 1) % capacity;
        size--;
    }
```

```
    // Display the queue elements
    void display() const {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return;
        }
        for (int i = 0; i < size; ++i) {
            cout << queue[(front + i) % capacity] << " ";
        }
        cout << endl;
    }
};

int main() {
    Queue q(5);

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.enqueue(50);

    q.display();

    q.dequeue();
    q.dequeue();

    q.display();

    cout << "Front element: " << q.frontElement() << endl;
    cout << "Back element: " << q.backElement() << endl;

    q.clear();
    q.display();

    return 0;
}
```



7. Supplementary Activity

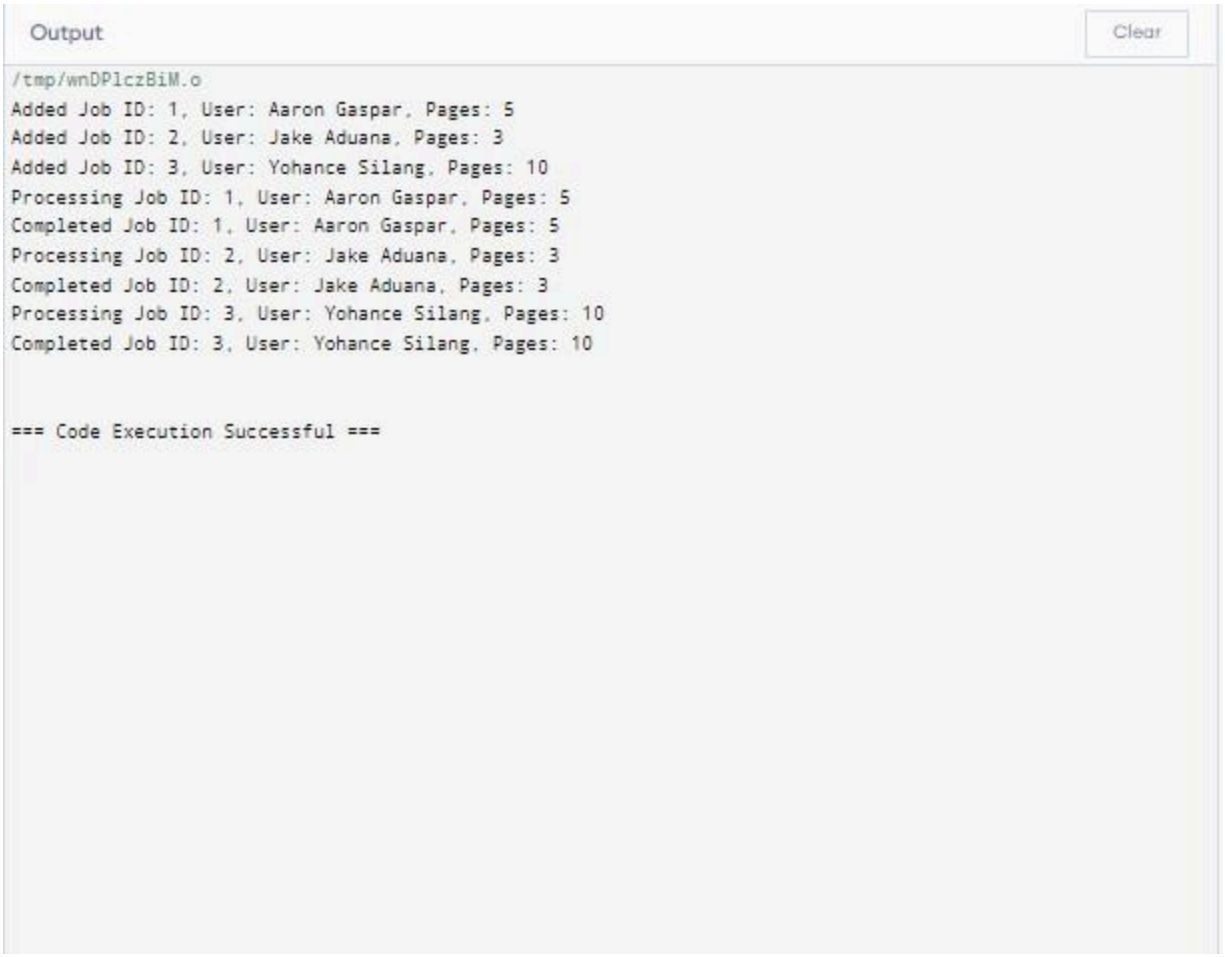
main.cpp

Share Run

```cpp
#include <iostream>
#include <string>
#include <thread>
#include <chrono>

const int MAX_JOBS = 100;  // Maximum number of jobs the printer can handle

class Job {
public:
    int job_id;
    std::string user_name;
    int pages;

    Job(int id, std::string name, int num_pages) : job_id(id), user_name(name), pages(num_pages)
        {}

    void printJob() const {
        std::cout << "Job ID: " << job_id << ", User: " << user_name << ", Pages: " << pages <<
            std::endl;
    }
};

class Printer {
private:
    Job* job_array[MAX_JOBS];
    int front;
    int rear;
    int count;

public:
    Printer() : front(0), rear(0), count(0) {}

    ~Printer() {
        for (int i = 0; i < count; ++i) {
            delete job_array[(front + i) % MAX_JOBS];
        }
    }
```

```
36
37      void addJob(Job* job) {
38          if (count == MAX_JOBS) {
39              std::cout << "Queue is full. Cannot add more jobs." << std::endl;
40              return;
41          }
42          job_array[rear] = job;
43          rear = (rear + 1) % MAX_JOBS;
44          count++;
45          std::cout << "Added ";
46          job->printJob();
47      }
48
49      void processJobs() {
50          while (count > 0) {
51              Job* current_job = job_array[front];
52              front = (front + 1) % MAX_JOBS;
53              count--;
54              std::cout << "Processing ";
55              current_job->printJob();
56              std::this_thread::sleep_for(std::chrono::seconds(current_job->pages * 2));  //
                    Simulate time taken to print
57              std::cout << "Completed ";
58              current_job->printJob();
59              delete current_job;
60          }
61      }
62  };
```

```
64  int main() {
65      Printer printer;
66
67      // Adding jobs to the printer
68      printer.addJob(Job(1, "Aaron Gaspar", 5));
69      printer.addJob(Job(2, "Jake Aduana", 3));
70      printer.addJob(Job(3, "Yohance Silang", 10));
71
72      // Processing the jobs
73      printer.processJobs();
74
75      return 0;
76  }
```



Using an array for the printer queue strikes a perfect balance between simplicity, efficiency, and predictability. It's a practical solution for environments where the maximum number of jobs is known and fixed, ensuring smooth and reliable operation. Arrays are one of the most fundamental data structures in programming. They are straightforward to understand and use, which makes them an excellent choice for implementing a queue. When you use an array, you don't have to worry about complex pointer manipulations or memory management issues that come with more advanced data structures like linked lists. This simplicity translates to fewer bugs and easier maintenance. When you use an array, you allocate a fixed amount of memory upfront. This predictability is crucial in environments where memory resources are limited or where you need to ensure that your application doesn't exceed a certain memory footprint. For example, in an embedded system or a real-time application, knowing exactly how much memory your queue will use can help you avoid unexpected crashes or performance issues.

8. Conclusion

In conclusion, learning queues in C++ can help a lot in terms of developing a program that requires queuing something. These techniques are the foundation to better development of skills in terms of practical setting. The values that it has will lead you to a path where you can compromise every problem that requires certain skill to accomplish. Every lesson has its purpose to its own and should be treated as equal as we treat ourselves.

9. Assessment Rubric