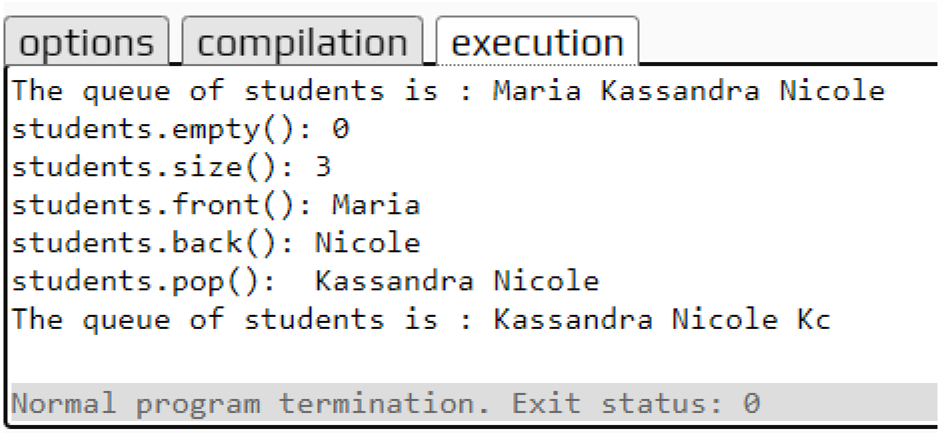
Activity No. 5	
QUEUES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/07/24
Section: CPE21S4	**Date Submitted:** 10/07/24
Name(s): Magistrado, Aira Pauleen M.	**Instructor:** Maria Rizette Sayo
6. Output	

C/C++

```cpp
// C++ code to illustrate queue in Standard Template Library (STL)
#include <iostream>
#include <queue>
#include <string>

//displays the queue
void display(std::queue<std::string> q) { //Copy the queue
    std::queue<std::string> c = q;
    while (!c.empty()) {
        std::cout << " " << c.front(); //access the element at front
        c.pop(); //it removes the front element after it is displayed
    }
    std::cout << "\n";
}

int main() {
    std::queue<std::string> students; //creates queue
    //push names to the queue
    students.push("Maria");
    students.push("Kassandra");
    students.push("Nicole");

    //outputs the queue of students
    std::cout << "The queue of students is :";
    display(students);

    std::cout << "students.empty(): " << students.empty() << "\n"; //checks if students
queue is empty
    std::cout << "students.size(): " << students.size() << "\n"; //checks the size
    std::cout << "students.front(): " << students.front() << "\n"; //checks student at
front
    std::cout << "students.back(): " << students.back() << "\n"; //checks student at back

    std::cout << "students.pop(): "; // remove the student at front
    students.pop();
    display(students);

    students.push("Kc"); //puts a new student in the queue
    std::cout << "The queue of students is :";
    display(students);
    return 0;
}
```

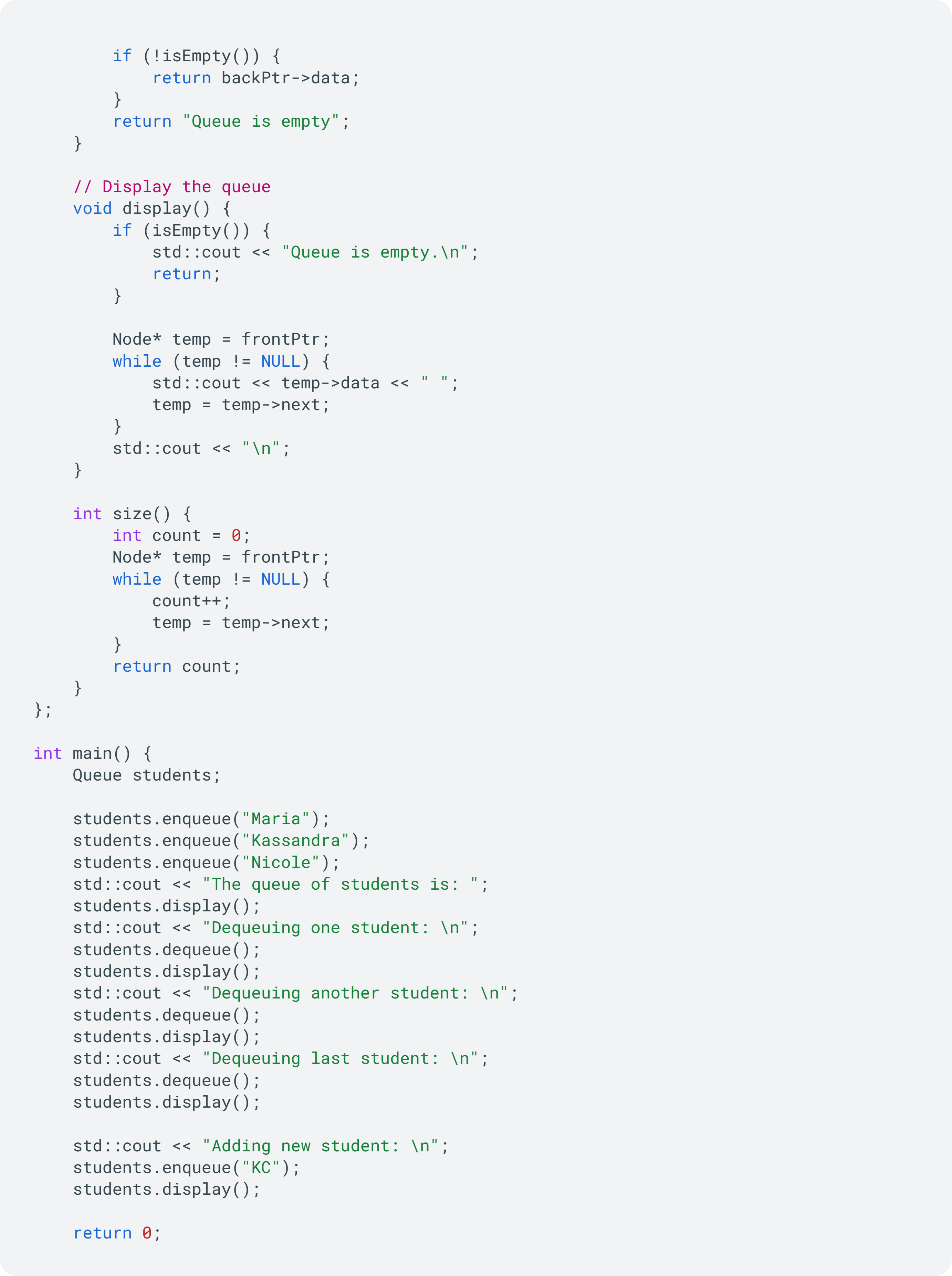


```
The queue of students is : Maria Kassandra Nicole
students.empty(): 0
students.size(): 3
students.front(): Maria
students.back(): Nicole
students.pop():  Kassandra Nicole
The queue of students is : Kassandra Nicole Kc

Normal program termination. Exit status: 0
```

Table 5-1. Queues using C++ STL

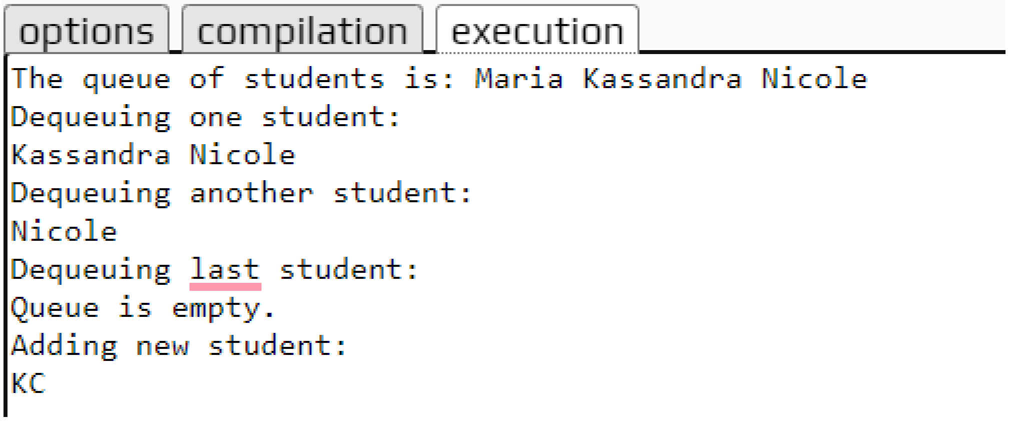
C/C++

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

class Node {
public:
    std::string data;
    Node* next;

    Node(std::string val) : data(val), next(NULL) {}
};

class Queue {
private:
    Node* frontPtr;
    Node* backPtr;

public:
    Queue() : frontPtr(NULL), backPtr(NULL) {}

    // Check if the queue is empty
    bool isEmpty() {
        return frontPtr == NULL;
    }

    // Insert an item
    void insertNonEmpty(std::string value) {
        Node* newPtr = new Node(value);
        newPtr->next = NULL;
        backPtr->next = newPtr;
        backPtr = newPtr;
    }
    void insertEmpty(std::string value) {
        Node* newPtr = new Node(value);
        frontPtr = newPtr;
        backPtr = newPtr;
    }

    // Enqueues operation
    void enqueue(std::string value) {
        if (isEmpty()) {
            insertEmpty(value);  // Insert to empty queue
        } else {
            insertNonEmpty(value);  //Insert to non-empty queue
        }
    }

    void deleteMoreThanOne() {
        Node* tempPtr = frontPtr;
        frontPtr = frontPtr->next;
        tempPtr->next = NULL;
        delete tempPtr;
    }

    void deleteOne() {
        Node* tempPtr = frontPtr;
        frontPtr = NULL;
        backPtr = NULL;
        delete tempPtr;
    }

    // Dequeue operation
    void dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty, can't dequeue.\n";
            return;
        }

        if (frontPtr == backPtr) {
            deleteOne();
        } else {
            deleteMoreThanOne();
        }
    }

    // front
    std::string front() {
        if (!isEmpty()) {
            return frontPtr->data;
        }
        return "Queue is empty";
    }

    // back
    std::string back() {
```



```cpp
        if (!isEmpty()) {
            return backPtr->data;
        }
        return "Queue is empty";
    }

    // Display the queue
    void display() {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
        }

        Node* temp = frontPtr;
        while (temp != NULL) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << "\n";
    }

    int size() {
        int count = 0;
        Node* temp = frontPtr;
        while (temp != NULL) {
            count++;
            temp = temp->next;
        }
        return count;
    }
};

int main() {
    Queue students;

    students.enqueue("Maria");
    students.enqueue("Kassandra");
    students.enqueue("Nicole");
    std::cout << "The queue of students is: ";
    students.display();
    std::cout << "Dequeuing one student: \n";
    students.dequeue();
    students.display();
    std::cout << "Dequeuing another student: \n";
    students.dequeue();
    students.display();
    std::cout << "Dequeuing last student: \n";
    students.dequeue();
    students.display();

    std::cout << "Adding new student: \n";
    students.enqueue("KC");
    students.display();

    return 0;
```

```
}
```

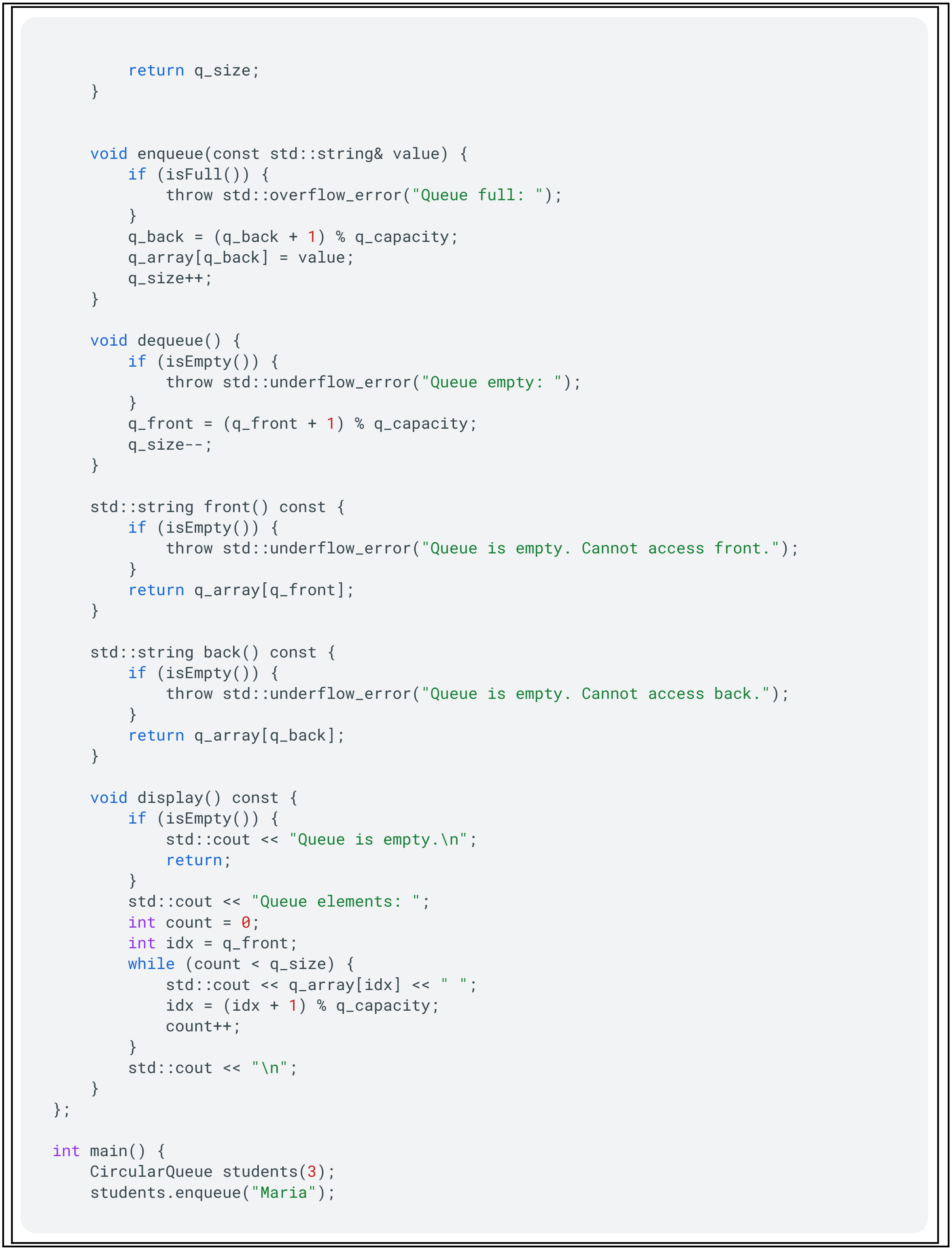


```
The queue of students is: Maria Kassandra Nicole
Dequeuing one student:
Kassandra Nicole
Dequeuing another student:
Nicole
Dequeuing last student:
Queue is empty.
Adding new student:
KC
```

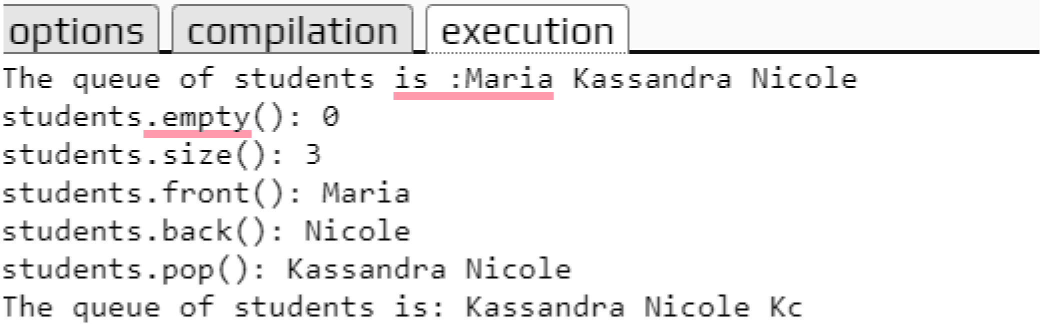
Table 5-2. Queues using Linked List Implementation

C/C++

```
#include <iostream>
#include <stdexcept>
#include <string>

class CircularQueue {
private:
    std::string* q_array;
    int q_capacity;
    int q_size;
    int q_front;
    int q_back;

public:
    CircularQueue(int capacity) : q_capacity(capacity), q_size(0), q_front(0),
q_back(q_capacity - 1) {
        q_array = new std::string[q_capacity];
    }

    ~CircularQueue() {
        delete[] q_array;
    }

    bool isEmpty() const {
        return q_size == 0;
    }

    bool isFull() const {
        return q_size == q_capacity;
    }

    int size() const {
```



```cpp
        return q_size;
    }


    void enqueue(const std::string& value) {
        if (isFull()) {
            throw std::overflow_error("Queue full: ");
        }
        q_back = (q_back + 1) % q_capacity;
        q_array[q_back] = value;
        q_size++;
    }

    void dequeue() {
        if (isEmpty()) {
            throw std::underflow_error("Queue empty: ");
        }
        q_front = (q_front + 1) % q_capacity;
        q_size--;
    }

    std::string front() const {
        if (isEmpty()) {
            throw std::underflow_error("Queue is empty. Cannot access front.");
        }
        return q_array[q_front];
    }

    std::string back() const {
        if (isEmpty()) {
            throw std::underflow_error("Queue is empty. Cannot access back.");
        }
        return q_array[q_back];
    }

    void display() const {
        if (isEmpty()) {
            std::cout << "Queue is empty.\n";
            return;
        }
        std::cout << "Queue elements: ";
        int count = 0;
        int idx = q_front;
        while (count < q_size) {
            std::cout << q_array[idx] << " ";
            idx = (idx + 1) % q_capacity;
            count++;
        }
        std::cout << "\n";
    }
};

int main() {
    CircularQueue students(3);
    students.enqueue("Maria");
    students.enqueue("Kassandra");
    students.enqueue("Nicole");

    std::cout << "The queue of students is :";
    students.display();


    std::cout << "students.empty(): " << students.isEmpty() << "\n"; // Checks if
students empty
    std::cout << "students.size(): " << students.size() << "\n";     // Checks size
    std::cout << "students.front(): " << students.front() << "\n";   // Checks student at
front
    std::cout << "students.back(): " << students.back() << "\n";     // Checks student at
back

    // Dequeue the student at front
    std::cout << "students.pop(): ";
    students.dequeue();
    students.display();

    // Push a new student into the queue
    students.enqueue("Kc");
    std::cout << "The queue of students is :";
    students.display();

    return 0;
}
```

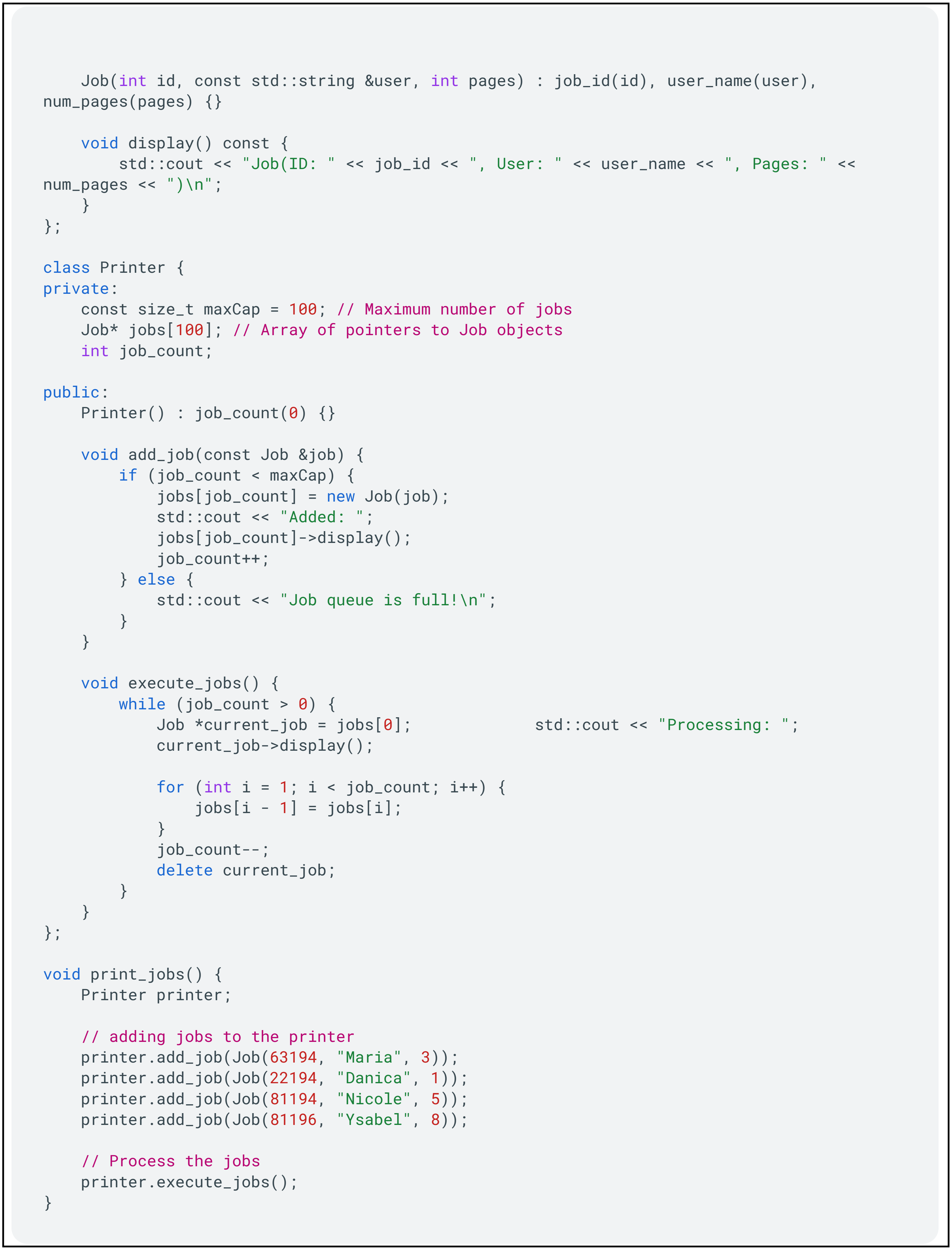


```
The queue of students is :Maria Kassandra Nicole
students.empty(): 0
students.size(): 3
students.front(): Maria
students.back(): Nicole
students.pop(): Kassandra Nicole
The queue of students is: Kassandra Nicole Kc
```

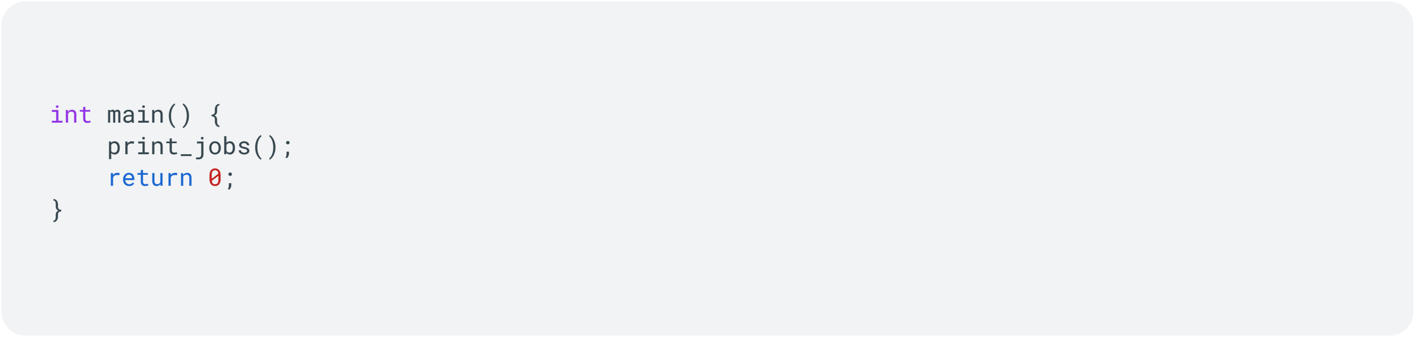
Table 5-3. Queues using Array Implementation

7. Supplementary Activity

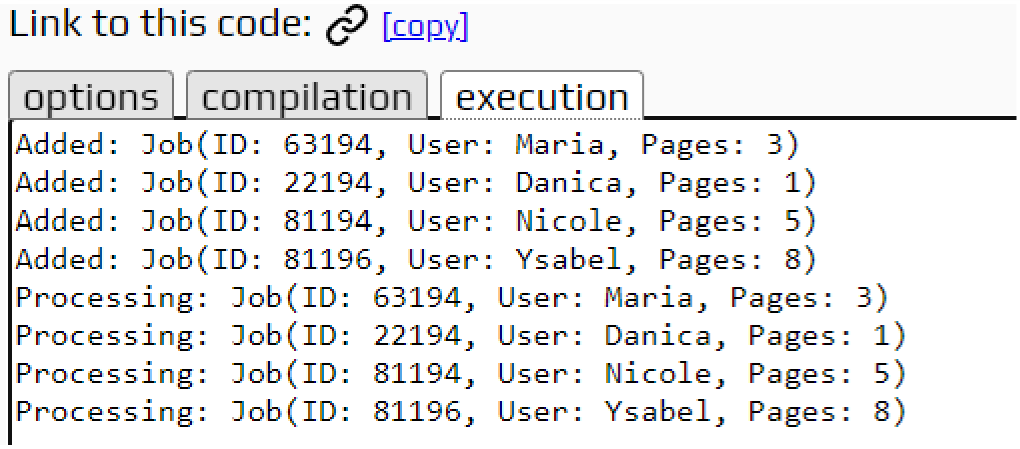
```
C/C++
#include <iostream>
#include <string>

class Job {
public:
    int job_id;
    std::string user_name;
    int num_pages;
```



```
    Job(int id, const std::string &user, int pages) : job_id(id), user_name(user),
num_pages(pages) {}

    void display() const {
        std::cout << "Job(ID: " << job_id << ", User: " << user_name << ", Pages: " <<
num_pages << ")\n";
    }
};

class Printer {
private:
    const size_t maxCap = 100; // Maximum number of jobs
    Job* jobs[100]; // Array of pointers to Job objects
    int job_count;

public:
    Printer() : job_count(0) {}

    void add_job(const Job &job) {
        if (job_count < maxCap) {
            jobs[job_count] = new Job(job);
            std::cout << "Added: ";
            jobs[job_count]->display();
            job_count++;
        } else {
            std::cout << "Job queue is full!\n";
        }
    }

    void execute_jobs() {
        while (job_count > 0) {
            Job *current_job = jobs[0];             std::cout << "Processing: ";
            current_job->display();

            for (int i = 1; i < job_count; i++) {
                jobs[i - 1] = jobs[i];
            }
            job_count--;
            delete current_job;
        }
    }
};

void print_jobs() {
    Printer printer;

    // adding jobs to the printer
    printer.add_job(Job(63194, "Maria", 3));
    printer.add_job(Job(22194, "Danica", 1));
    printer.add_job(Job(81194, "Nicole", 5));
    printer.add_job(Job(81196, "Ysabel", 8));

    // Process the jobs
    printer.execute_jobs();
}
```



```
int main() {
    print_jobs();
    return 0;
}
```



```
Added: Job(ID: 63194, User: Maria, Pages: 3)
Added: Job(ID: 22194, User: Danica, Pages: 1)
Added: Job(ID: 81194, User: Nicole, Pages: 5)
Added: Job(ID: 81196, User: Ysabel, Pages: 8)
Processing: Job(ID: 63194, User: Maria, Pages: 3)
Processing: Job(ID: 22194, User: Danica, Pages: 1)
Processing: Job(ID: 81194, User: Nicole, Pages: 5)
Processing: Job(ID: 81196, User: Ysabel, Pages: 8)
```

I used arrays because I find them easier to implement, understand, or use. It stores and organizes a fixed number of jobs and is straightforward compared to linked lists. The program begins by adding jobs to the array one at a time. It processes jobs in the sequence in which they were added (first job in, first job out) as it is a queue. After processing a job, the program forwards all remaining jobs in the array to keep the order

8. Conclusion

In conclusion, I have learned how to implement queues using array-based and linked list structures using queue operations such as enqueue, dequeue, and display in this activity. The array implementation is space-efficient but came with a limitation in size while the linked list could grow and shrink as needed. In the supplementary, I gained a deeper understanding of how queues can be applied to real-world problems like in printing jobs. I used an array because I find it easier and faster to implement than linked lists. I think I did a good job but I need to practice more to improve my skills and to understand coding better.

9. Assessment Rubric