

Activity No. 5	
QUEUES	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed: 10/7/2024
Section: CPE21S4	Date Submitted: 10/7/2024
Name(s): ALEXZANDER J. REYES	Instructor: Prof. MARIA RIZETTE SAYO

6. Output

```
main.cpp
1 #include <iostream>
2 #include <queue>
3 #include <string>
4
5 using namespace std;
6
7 // Function to display the contents of the queue
8 void display(queue<string> q) {
9     queue<string> temp = q;
10    while (!temp.empty()) {
11        cout << temp.front() << " ";
12        temp.pop();
13    }
14    cout << endl;
15 }
16
17 int main() {
18     // Declare a queue to store students' names
19     queue<string> students;
20
21     // Array of student names
22     string studentNames[] = {"Alice", "Bob", "Charlie", "David", "Eve"};
23
24     // Push each student name to the queue
25     for (int i = 0; i < 5; ++i) {
26         students.push(studentNames[i]);
27         cout << "Added: " << studentNames[i] << " to the queue.\n";
28     }
29
30     // Display the queue
31     cout << "Current queue: ";
32     display(students);
33
34     // Observing queue operations
35     cout << "Is the queue empty? : " << (students.empty() ? "Yes" : "No") << endl;
36     cout << "Size of the queue: " << students.size() << endl;
37     cout << "Front of the queue: " << students.front() << endl;
38     cout << "Back of the queue: " << students.back() << endl;
39
40     // Pop an element and observe the queue
41     cout << "Popping the front element: " << students.front() << endl;
42     students.pop();
43
44     // Display the queue after pop
45     cout << "Queue after pop: ";
46     display(students);
47
48     // Add another student to the queue
49     students.push("Frank");
50     cout << "Added: Frank to the queue.\n";
51
52     // Display the queue after adding a new element
53     cout << "Queue after adding Frank: ";
54     display(students);
55
56     return 0;
57 }
58
```

```
Output
/tmp/V6vR0xSVf9.o
Added: Alice to the queue.
Added: Bob to the queue.
Added: Charlie to the queue.
Added: David to the queue.
Added: Eve to the queue.
Current queue: Alice Bob Charlie David Eve
Is the queue empty? : No
Size of the queue: 5
Front of the queue: Alice
Back of the queue: Eve
Popping the front element: Alice
Queue after pop: Bob Charlie David Eve
Added: Frank to the queue.
Queue after adding Frank: Bob Charlie David Eve Frank

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

Table 5-1. Queues using C++ STL

**Observations:**

1. The queue is initially filled with student names in the order they are pushed.
2. After popping the front element, the next element (Bob) becomes the new front.
3. The size of the queue decreases by 1 after the pop operation.
4. Adding a new element (Frank) places it at the back of the queue, following the FIFO principle.

**INPUT:**

```
#include <iostream>

using namespace std;

// Node class representing each element in the queue
class Node {
public:
    int data;

    Node* next;

    // Constructor to initialize a new node
    Node(int val) : data(val), next(nullptr) {}
};

// Queue class implementing queue operations using a linked list
class Queue {
private:
    Node* front;

    Node* rear;
public:
    // Constructor to initialize an empty queue
    Queue() : front(nullptr), rear(nullptr) {}

    // Check if the queue is empty
```

```

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

// Insert an item into the queue (both empty and non-empty cases)
void enqueue(int value) {
    Node* newNode = new Node(value);

    // If the queue is empty, set both front and rear to the new node
    if (isEmpty()) {
        front = rear = newNode;

        cout << "Inserted " << value << " into an empty queue.\n";
    } else {
        // If the queue is non-empty, add the new node at the rear
        rear->next = newNode;
        rear = newNode;

        cout << "Inserted " << value << " into a non-empty queue.\n";
    }
}

// Delete an item from the queue (both cases: one item and more than one item)
void dequeue() {
    if (isEmpty()) {
        cout << "Queue is empty, cannot dequeue.\n";
        return;
    }

    // If the queue has only one item
    if (front == rear) {
        cout << "Deleted " << front->data << " from a queue with one item.\n";
        delete front;
    }
}

```

```

    front = rear = nullptr;

} else {

    // If the queue has more than one item

    Node* temp = front;

    front = front->next;

    cout << "Deleted " << temp->data << " from a queue of more than one item.\n";

    delete temp;

}

}

// Display the contents of the queue

void display() {

    if (isEmpty()) {

        cout << "Queue is empty.\n";

        return;

    }

    Node* temp = front;

    while (temp != nullptr) {

        cout << temp->data << " ";

        temp = temp->next;

    }

    cout << endl;

}

// Destructor to clear the queue

~Queue() {

    while (!isEmpty()) {

        dequeue();

    }

}

```

```
}  
};  
// Main function to test the queue operations  
int main() {  
    Queue q;  
    // Inserting into an empty queue  
    q.enqueue(10); // Expected: Insert into an empty queue  
    q.display();  
    // Inserting into a non-empty queue  
    q.enqueue(20); // Expected: Insert into a non-empty queue  
    q.enqueue(30); // Expected: Insert into a non-empty queue  
    q.display();  
    // Deleting from a queue of more than one item  
    q.dequeue(); // Expected: Deleting from queue of more than one item  
    q.display();  
    // Deleting from a queue with one item  
    q.dequeue(); // Expected: Deleting from queue of more than one item  
    q.dequeue(); // Expected: Deleting from queue with one item  
    q.display();  
    // Try to delete from an empty queue  
    q.dequeue(); // Expected: Queue is empty, cannot dequeue  
    return 0;  
}
```

```
Output
/tmp/evWQ5n2h6.o
Inserted 10 into an empty queue.
10
Inserted 20 into a non-empty queue.
Inserted 30 into a non-empty queue.
10 20 30
Deleted 10 from a queue of more than one item.
20 30
Deleted 20 from a queue of more than one item.
Deleted 30 from a queue with one item.
Queue is empty.
Queue is empty, cannot dequeue.

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

Table 5-2. Queues using Linked List Implementation

**Observations:**

Inserting into an empty queue:

The first element (10) is inserted and becomes both the front and rear of the queue.

Inserting into a non-empty queue:

Subsequent elements (20 and 30) are added to the rear of the queue. The rear pointer is updated each time.

Deleting from a queue of more than one item:

When deleting the first item (10), the front moves to the next node (20), and 10 is removed. The queue still has more elements.

Deleting from a queue with one item:

After deleting 20, there is only one item left (30). When this item is deleted, both front and rear are set to nullptr, indicating that the queue is now empty.

```
#include <iostream>
```

```
using namespace std;
```

```
class Queue {
private:
    int front;
    int rear;
```

```
int capacity; // Capacity of the array (size limit)
int* arr;    // Array to store the elements
int count;  // Current number of elements in the queue
```

public:

```
// Constructor to initialize the queue
```

```
Queue(int size) {
    front = 0;
    rear = -1;
    capacity = size;
    count = 0;
    arr = new int[capacity];
}
```

```
// Destructor to delete the allocated array
```

```
~Queue() {
    delete[] arr;
}
```

```
// Function to add an element to the queue (enqueue)
```

```
void enqueue(int value) {
    if (isFull()) {
        cout << "Queue overflow. Cannot insert " << value << ".\n";
        return;
    }
    rear = (rear + 1) % capacity; // Circular increment
    arr[rear] = value;
    count++;
    cout << "Inserted " << value << " into the queue.\n";
}
```

```
// Function to remove the front element from the queue (dequeue)
```

```
void dequeue() {
    if (isEmpty()) {
        cout << "Queue underflow. No element to dequeue.\n";
        return;
    }
    cout << "Removed " << arr[front] << " from the queue.\n";
    front = (front + 1) % capacity; // Circular increment
    count--;
}
```

```
// Function to get the front element of the queue
```

```
int getFront() {
    if (isEmpty()) {
        cout << "Queue is empty.\n";
        return -1;
    }
    return arr[front];
}
```

```

// Function to check if the queue is empty
bool isEmpty() {
    return (count == 0);
}

// Function to check if the queue is full
bool isFull() {
    return (count == capacity);
}

// Function to display the contents of the queue
void display() {
    if (isEmpty()) {
        cout << "Queue is empty.\n";
        return;
    }
    cout << "Queue contents: ";
    for (int i = 0; i < count; i++) {
        int index = (front + i) % capacity; // Circular indexing
        cout << arr[index] << " ";
    }
    cout << endl;
}

// Function to get the size of the queue
int size() {
    return count;
}
};

int main() {
    // Create a queue with a capacity of 5
    Queue q(5);

    // Inserting elements into the queue
    q.enqueue(10); // Insert 10
    q.enqueue(20); // Insert 20
    q.enqueue(30); // Insert 30
    q.enqueue(40); // Insert 40
    q.enqueue(50); // Insert 50
    q.display(); // Show queue

    // Trying to insert an element into a full queue
    q.enqueue(60); // Should indicate queue overflow

    // Display front element
    cout << "Front element: " << q.getFront() << endl;

    // Remove elements
    q.dequeue(); // Remove front element (10)
    q.display(); // Show queue after removal
}

```



```

q.dequeue(); // Remove front element (20)
q.display(); // Show queue after removal

// Insert new element
q.enqueue(60); // Insert 60 into queue
q.display(); // Show queue after insertion

return 0;
}

```

Output

```

/tmp/EyKJpUMCi5.o
Inserted 10 into the queue.
Inserted 20 into the queue.
Inserted 30 into the queue.
Inserted 40 into the queue.
Inserted 50 into the queue.
Queue contents: 10 20 30 40 50
Queue overflow. Cannot insert 60.
Front element: 10
Removed 10 from the queue.
Queue contents: 20 30 40 50
Removed 20 from the queue.
Queue contents: 30 40 50
Inserted 60 into the queue.
Queue contents: 30 40 50 60

=== Code Execution Successful ===

```

Table 5-3. Queues using Array Implementation

#### Observations:

Insertion into a Full Queue:

When trying to insert an element into a full queue, the program detects the overflow and prevents the operation.

Circular Array Management:

The queue uses circular indexing, ensuring that elements can be added and removed in a circular fashion, without shifting elements, improving efficiency.

Queue Operations:

The queue properly handles insertion (enqueue) and removal (dequeue) while maintaining the correct order of elements, as expected from a queue's FIFO behavior.

## 7. Supplementary Activity

### ILO C: Solve problems using queue implementation

**Problem Title:** Shared Printer Simulation using Queues

**Problem Definition:** In this activity, we'll simulate a queue for a shared printer in an office. In any corporate office, usually, the printer is shared across the whole floor in the printer room. All the computers in this room are connected to the same printer. But a printer can do only one printing job at any point in time, and it also takes some time to complete any job. In the meantime, some other user can send another print request. In such a case, a printer needs to store all the pending jobs somewhere so that it can take them up once its current task is done.

Perform the following steps to solve the activity. **Make sure that you include a screenshot of the source code**

**for each.** From the get-go: You must choose whether you are making a linked list or array implementation.

**You may NOT use the STL.**

1. Create a class called Job (comprising an ID for the job, the name of the user who submitted it, and the number of pages).
2. Create a class called Printer. This will provide an interface to add new jobs and process all the jobs added so far.
3. To implement the printer class, it will need to store all the pending jobs. We'll implement a very basic strategy – first come, first served. Whoever submits the job first will be the first to get the job done.
4. Finally, simulate a scenario where multiple people are adding jobs to the printer, and the printer is processing them one by one.
5. Defend your choice of internal representation: Why did you use arrays or linked list?

### Output Analysis:

- Provide the output after performing each task above.
- Include your analysis: focus on why you think the output is the way it is.

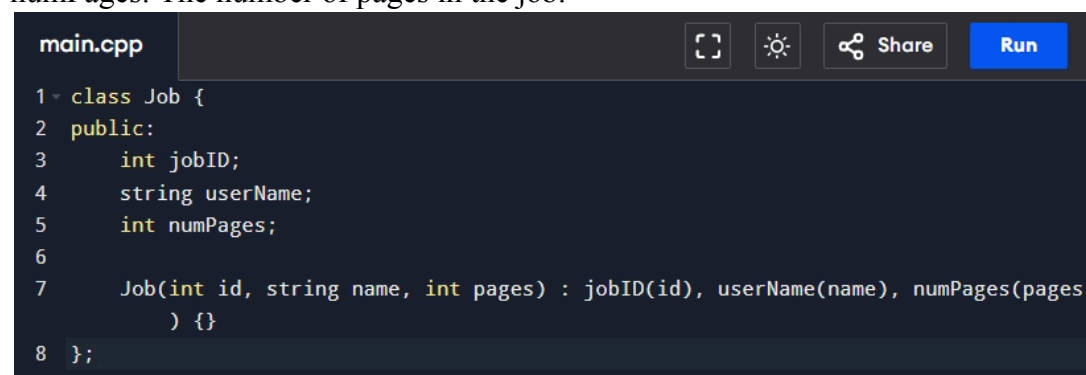
#### 1. Create a class called Job

The Job class contains the following attributes:

jobID: The ID of the job.

userName: The name of the user who submitted the job.




numPages: The number of pages in the job.



```
main.cpp  [Icons] Share Run
1 class Job {
2 public:
3     int jobID;
4     string userName;
5     int numPages;
6
7     Job(int id, string name, int pages) : jobID(id), userName(name), numPages(pages) {}
8 };
```

## 2. Create a class called Printer

The Printer class provides an interface to add new jobs and process the jobs. It maintains the queue of print jobs and processes them in a first-come, first-served manner.

```
main.cpp    Share





1- class Printer {
2- private:
3-     Node* front; // Points to the front of the queue
4-     Node* rear;  // Points to the rear of the queue
5-
6- public:
7-     Printer() : front(nullptr), rear(nullptr) {}
8-
9-     // Add job to the printer queue
10-    void addJob(Job* newJob) {
11-        Node* newNode = new Node(newJob);
12-        if (rear == nullptr) {
13-            front = rear = newNode;
14-        } else {
15-            rear->next = newNode;
16-            rear = newNode;
17-        }
18-        cout << "Job ID " << newJob->jobID << " added by " << newJob->userName
19-              << " with " << newJob->numPages << " pages.\n";
20-    }
21-
22-    // Process all jobs in the queue
23-    void processJobs() {
24-        while (front != nullptr) {
25-            cout << "Processing Job ID " << front->job->jobID << " for "
26-                  << front->job->userName << " with " << front->job->numPages << " pages...\n";
27-
28-            // Simulate job processing (print)
29-            Node* temp = front;
30-            front = front->next;
31-            delete temp;
32-        }
33-        rear = nullptr;
34-        cout << "All jobs processed.\n";
35-    }
36-};
```

## 3. Implement First-Come, First-Served Strategy

The printer class processes jobs in the order they were added to the queue, following the First-Come, First-Served (FIFO) strategy. This ensures that the first job added is processed before any others.

## 4. Simulate a Scenario with Multiple Users

In the simulation, multiple users add print jobs, and the printer processes them one by one in the order they were added.

```
main.cpp    Share  Run

1- int main() {
2-     Printer printer;
3-
4-     // Adding jobs to the printer queue
5-     printer.addJob(new Job(1, "Alice", 10));
6-     printer.addJob(new Job(2, "Bob", 5));
7-     printer.addJob(new Job(3, "Charlie", 20));
8-
9-     // Processing all jobs in the queue
10-    printer.processJobs();
11-
12-    return 0;
13- }
14-
```

## 5. Defend Your Choice of Internal Representation

**Dynamic Size:** A linked list allows the queue to grow and shrink dynamically as jobs are added and processed. This avoids the need to resize the structure, which is necessary in arrays when their capacity is exceeded.

**Efficient Operations:** In a linked list, adding jobs to the rear of the queue and processing (removing) jobs from the front both take  $O(1)$  time. In contrast, an array-based queue might require costly resizing operations when full.

**Memory Efficiency:** A linked list uses memory only for the jobs currently in the queue, whereas an array may allocate extra memory even if the queue has fewer elements.

Thus, the linked list is well-suited for this scenario, where the number of pending jobs can vary, and we need efficient insertion and deletion of jobs.

### OUTPUT:

```
Output
/tmp/e1NOTcltVA.o
Job ID 1 added by Alice with 10 pages.
Job ID 2 added by Bob with 5 pages.
Job ID 3 added by Charlie with 20 pages.
Processing Job ID 1 for Alice with 10 pages...
Processing Job ID 2 for Bob with 5 pages...
Processing Job ID 3 for Charlie with 20 pages...
All jobs processed.

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

#### Job Addition Output:

The jobs are added sequentially to the queue as per the `addJob()` function. The output shows each job's ID, the name of the user submitting it, and the number of pages, confirming that each job has been enqueued successfully. The order of the jobs in the queue reflects the order in which they were submitted.

**Reason:** This behavior is due to the implementation of the linked-list-based queue where each job is added to the rear (end) of the queue. This ensures that jobs maintain the order in which they are submitted.

#### Job Processing Output:

The jobs are processed one by one in the exact order they were added to the queue, starting with Alice's job (Job ID 1) and ending with Charlie's job (Job ID 3). After processing each job, the queue is dequeued from the front. When all jobs are processed, the system outputs a message stating that all jobs have been completed.

**Reason:** The jobs are processed in a First-Come, First-Served (FIFO) manner. The `processJobs()` function dequeues each job from the front of the queue, ensuring the oldest job is processed first. This is the expected behavior of a queue data structure. After processing all jobs, the queue is empty, and the message "All jobs processed" confirms that the queue has been cleared.

## 8. Conclusion

In this activity, we successfully implemented a queue using both linked lists and arrays, exploring key operations such as insertion, deletion, and traversal. The linked list implementation allowed dynamic memory management and handled insertion/deletion efficiently without shifting elements. In contrast, the array implementation, while simpler, required circular indexing to manage space limitations and prevent overflow. Both implementations adhered to the First-In-First-Out (FIFO) principle, and the simulations demonstrated their effectiveness for managing and processing data in a queue structure.

## 9. Assessment Rubric