Queues; In Depth in C++

By: Aaryan Vinod Kumar

In the previous lesson, we learned about Queues; a linear data structure following the FIFO (First In, First Out) principle.

However, the basic queue has certain limitations:

- It doesn't prioritize tasks all elements are treated equally.
- It can waste memory in array implementations after multiple enqueue/dequeue operations.

To overcome these limitations, we use Priority Queues and Circular Queues.

What is a Priority Queue?

A **Priority Queue** is a special type of queue where **each element is assigned a priority**.
Instead of following strict FIFO order, the **element with the highest priority is dequeued first**, regardless of insertion order.

If two elements have the same priority, the one that appears first (based on insertion order) is served first.

How It Works:

- Elements are arranged based on **priority**, not insertion time.
- Internally, it's often implemented using a **heap** (a binary heap in STL).

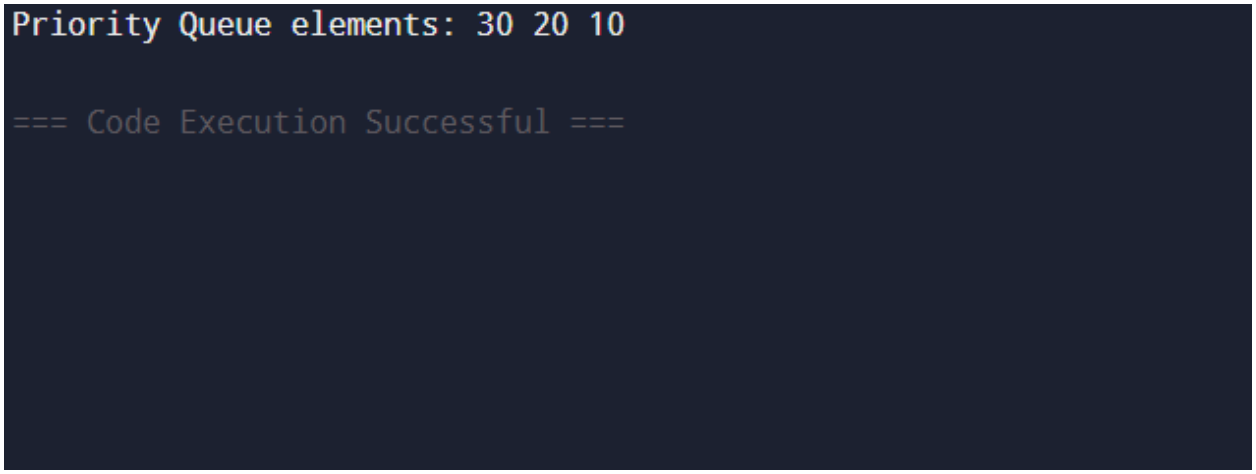
Common Priority Queue Functions:

Function	Description
pq.push(x)	Inserts element x into the queue
pq.pop()	Removes the highest priority element
pq.top()	Returns the element with highest priority
pq.empty()	Checks if queue is empty
pq.size()	Returns number of elements

Sample Program For Priority Queue:

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

int main() {
    priority_queue<int> pq;
    pq.push(10);
    pq.push(30);
    pq.push(20);

    cout << "Priority Queue elements: ";
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }

    return 0;
}
```

Output:


Priority Queue elements: 30 20 10
=== Code Execution Successful ===

Operations on Priority Queue

Operation	Time Complexity
Insertion (push)	O(log n)
Deletion (pop)	O(log n)
Access Top	O(1)
Search	O(n)

Real-World Use Cases

- CPU process scheduling (higher priority tasks first)
- Dijkstra’s algorithm (shortest path)
- Event-driven simulation systems
- Job scheduling and networking packets

What is a Circular Queue?

A **Circular Queue** is a modified version of a linear queue where the **last position is connected back to the first position**, forming a **circle**.

This prevents the problem of wasted space that occurs in a simple array-based queue after repeated dequeues.

How It Works:

- The rear pointer moves circularly using (rear + 1) % size.
- The front pointer moves in the same circular fashion.
- The queue is full when (rear + 1) % size == front.
- The queue is empty when front == -1.

Sample Program For Circular Queue:

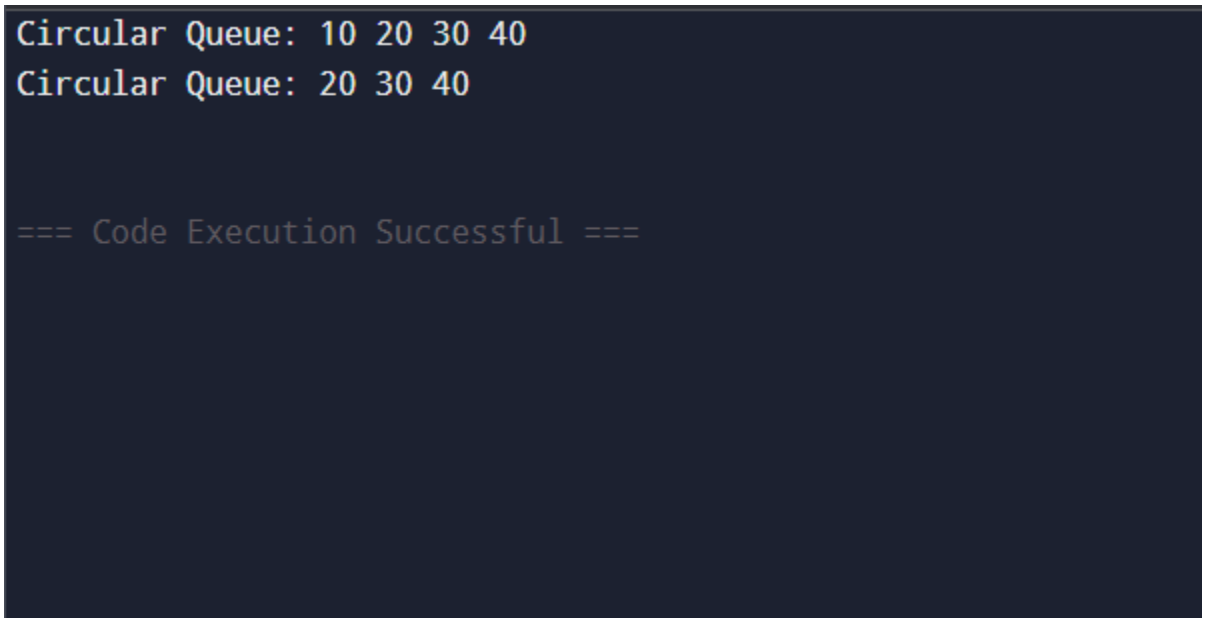
```
#include <iostream>

using namespace std;


#define SIZE 5


class CircularQueue {

    int items[SIZE], front, rear;

public:

    CircularQueue() {

        front = -1;

        rear = -1;

    }


    bool isFull() {

        return (front == 0 && rear == SIZE - 1) || (rear + 1 ==
front);

    }


    bool isEmpty() {

        return front == -1;

    }
```

```
void enqueue(int element) {
    if (isFull()) {
        cout << "Queue is full\n";
        return;
    }
    if (front == -1) front = 0;
    rear = (rear + 1) % SIZE;
    items[rear] = element;
}

void dequeue() {
    if (isEmpty()) {
        cout << "Queue is empty\n";
        return;
    }
    if (front == rear) {
        front = -1;
        rear = -1;
    } else {
        front = (front + 1) % SIZE;
```

```
        }
    }

    void display() {
        if (isEmpty()) {
            cout << "Queue is empty\n";
            return;
        }
        cout << "Circular Queue: ";
        int i = front;
        while (true) {
            cout << items[i] << " ";
            if (i == rear) break;
            i = (i + 1) % SIZE;
        }
        cout << endl;
    }
};

int main() {
    CircularQueue q;
```

```
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.display();
    q.dequeue();
    q.display();
}

//Don't stress over how long the code is, remember the concept
```

Output:



Operations on Circular Queue

Operation	Time Complexity
Enqueue	O(1)
Dequeue	O(1)
Access Front/Rear	O(1)

Real-World Use Cases

- Circular buffers in operating systems
- Real-time data streaming (audio/video buffers)
- Network traffic management