Queues

This C++ file is a perfect, side-by-side comparison of two array-based implementations of the **Queue data structure**, both neatly encapsulated within C++ classes. The file's primary lesson is showcasing the evolution from a simple, intuitive design to a highly efficient and standard one.

It begins with a straightforward **Queue** that uses two pointers, front and rear. While easy to understand, this implementation has a critical flaw: it wastes memory. Once an element is dequeued, the space it occupied at the front of the array can never be used again.

The file then presents the solution: the **Circular Queue**. This is the industry-standard way to implement a fixed-size queue. By cleverly using the **modulo operator (%)**, it treats the linear array as a circle. This allows the queue to "wrap around" and reuse the empty spaces at the beginning of the array, leading to a much more efficient use of memory. The use of C++ classes with constructors and destructors also ensures proper resource management for both implementations.

**Implementation of Queue using 2 pointers**

This section implements a basic **Queue** using a standard array. It follows the **First-In, First-Out (FIFO)** principle but suffers from a significant memory inefficiency. Once space at the front is cleared, it cannot be reused.

* class Queue{ ... private: ... public: ... }; This encapsulates the queue's data (size, front, rear, Q) and logic into a C++ class. The data members are kept **private** to prevent direct, unsafe manipulation, while the functions (enqueue, dequeue) provide a safe **public** interface.
* Queue::Queue(int size) { ... front = -1; rear = -1; ... } This is the **constructor**. It initializes a new queue object. Setting front and rear to -1 is a common convention to indicate that the queue is **empty**.
* bool Queue::isFull() { if (rear == size-1){ ... } } This isFull check is the source of the implementation's **inefficiency**. It only considers the queue full when the rear pointer reaches the physical end of the array. It doesn't account for any empty slots that might have opened up at the front after dequeuing elements.
* int Queue::dequeue() { ... front++; x = Q[front]; ... } The dequeue operation simply **increments the front pointer**. The old data is not erased, but the space it occupies is now considered "dead" and cannot be reclaimed by this simple implementation, leading to wasted memory.

**Circular Queue using array**

This is the highly efficient, standard implementation of an array-based queue. It solves the wasted-space problem of the simple queue by treating the array as a circle.

* CircularQueue::CircularQueue(int size) { ... front = 0; rear = 0; ... } In the circular queue, the pointers are initialized to **0**. The condition for an empty queue now becomes front == rear. To avoid ambiguity with a full queue, the array is typically created with one extra space (size + 1).
* bool CircularQueue::isFull() { if ((rear + 1) % size == front){ ... } } This is the correct and robust way to check if a circular queue is full. The **modulo operator (%)** calculates where the *next* rear position would be. If this new position collides with the current front, the queue is full.
* void CircularQueue::enqueue(int x) { ... rear = (rear + 1) % size; Q[rear] = x; } This is the core of the circular logic. When adding an element, the rear pointer is incremented and then the modulo is applied. This makes the pointer automatically **"wrap around"** to the beginning of the array if it was at the end, allowing it to reuse any empty slots.
* void CircularQueue::display() { ... do { ... } while (i != (rear + 1) % size); } Displaying a circular queue often requires a do-while loop. This ensures that the loop's body executes at least once, which is necessary to print all elements correctly, especially when the queue has wrapped around and the rear index is smaller than the front index.