Stacks and Queues in C++

By: Aaryan Vinod Kumar

In programming, Stacks and Queues are linear data structures meaning their elements are stored and accessed in a sequential manner. However, what makes them different is how elements are inserted and removed.

> A Stack follows the LIFO (Last In, First Out) principle; the last element added is the first one to be removed.
>
> A Queue follows the FIFO (First In, First Out) principle; the first element added is the first one to be removed.

Both are essential for managing data where order of access matters, such as tracking function calls, simulating waiting lines, or processing requests in order.

What is a Stack?

A **Stack** is a collection of elements that allows insertion and deletion **only from one end**, called the **top** of the stack.
You can think of it like a stack of plates; you add a plate on top and remove the topmost one first.

Basic Operations:

1. **Push:** Add an element to the top.

2. **Pop:** Remove the top element.

3. **Peek / Top:** View the element at the top without removing it.

4. **isEmpty:** Check if the stack has no elements.

What is a Queue?

A **Queue** is a collection of elements that allows insertion from one end (called the **rear**) and deletion from the other end (called the **front**).
It's like people standing in line, the first one to arrive gets served first.

Basic Operations:

1. **Enqueue:** Add an element to the rear.

2. **Dequeue:** Remove an element from the front.

3. **Front / Peek:** View the element at the front without removing it.

4. **isEmpty:** Check if the queue is empty.

Why Use Stacks and Queues?

- **Efficient Order Handling:**
  These structures naturally model many real-world processes involving order like call stacks or print queues.

- **Simplifies Algorithms:**
  Recursive functions, undo systems, and BFS/DFS traversal rely on these for organized data management.

- **Easy to Implement:**
  Can be built using arrays, linked lists, or directly used from the C++ STL.

- **Fast Operations:**
  Insertions and deletions take constant time **O(1)** since they happen only at one or two fixed ends.

Key Terms Explained

- LIFO (Last In, First Out):
  The most recent element added is removed first, used by *Stacks*.

- FIFO (First In, First Out):
  The earliest element added is removed first, used by *Queues*.

- Container Adaptor:
  A class that uses another container (like `deque` or `vector`) internally to provide a specific restricted interface.

- Overflow & Underflow:
  Overflow occurs when adding to a full stack/queue (in static implementations).
  Underflow occurs when trying to remove from an empty one.

Stack Syntax:

```
#include <stack>

using namespace std;


stack<int> s;   // Declaration
```

Common Stack Functions:

Function	Description
`s.push(x)`	Adds element x on top
`s.pop()`	Removes the top element
`s.top()`	Returns the top element
`s.empty()`	Returns true if empty
`s.size()`	Returns number of elements

Queue Syntax:

```
#include <queue>

using namespace std;


queue<int> q;   // Declaration
```

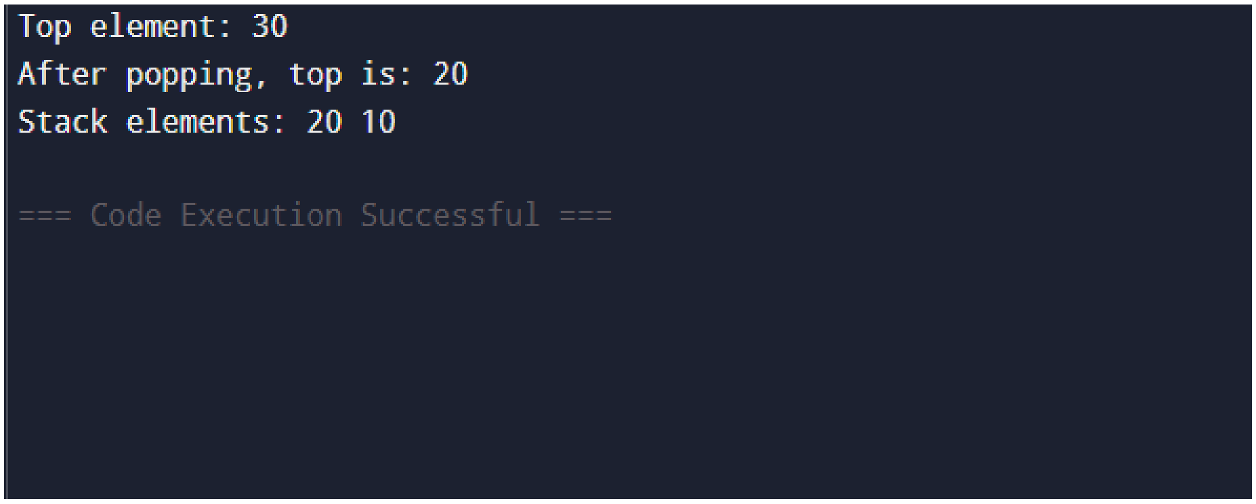
Common Queue Functions:

Function	Description
`q.push(x)`	Adds element x at the rear
`q.pop()`	Removes the front element
`q.front()`	Returns front element
`q.back()`	Returns last element
`q.empty()`	Checks if queue is empty
`q.size()`	Returns number of elements

Sample Program For Stack:

```
#include <iostream>

#include <stack>

using namespace std;

int main() {

    stack<int> s;

    s.push(10);

    s.push(20);

    s.push(30);


    cout << "Top element: " << s.top() << endl;

    s.pop();


    cout << "After popping, top is: " << s.top() << endl;

    cout << "Stack elements: ";

    while (!s.empty()) {

        cout << s.top() << " ";

        s.pop();

    }

    return 0;

}
```

Output:

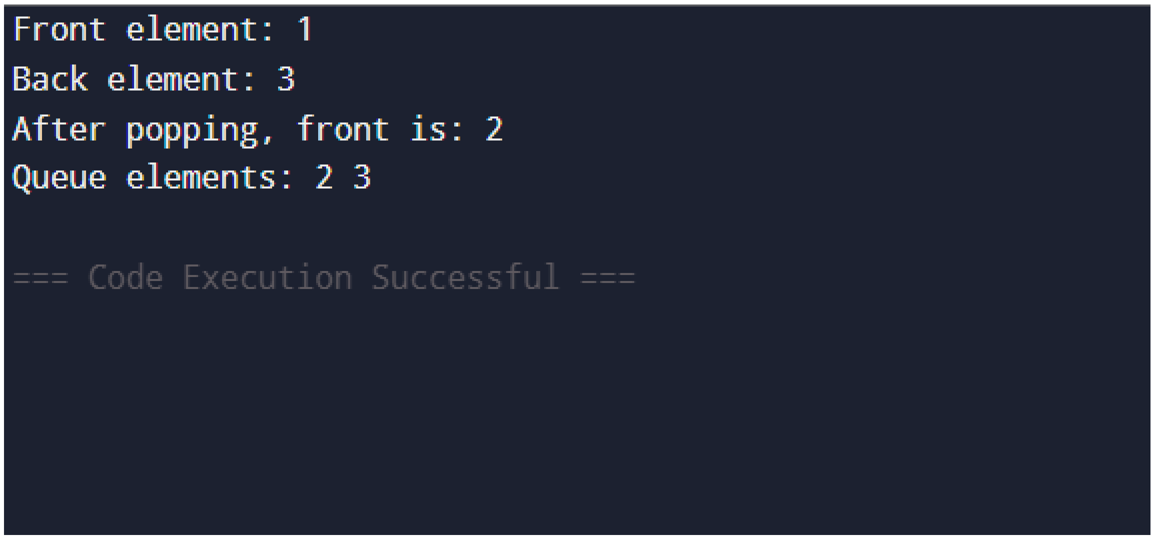


```
Top element: 30
After popping, top is: 20
Stack elements: 20 10

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

Sample Program For Stack:

```
#include <iostream>

#include <queue>

using namespace std;

int main() {

    queue<int> q;

    q.push(1);

    q.push(2);

    q.push(3);

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

    cout << "Back element: " << q.back() << endl;

    q.pop(); //Removing the last element

    cout << "After popping, front is: " << q.front() << endl;


    cout << "Queue elements: ";

    while (!q.empty()) {

        cout << q.front() << " ";

        q.pop();

    }

    return 0;

}
```

Output:



```
Front element: 1
Back element: 3
After popping, front is: 2
Queue elements: 2 3

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

Disadvantages

Stack:

- Only the top element is directly accessible.
- Not suitable for random access or traversal.
- In manual array implementations, overflow/underflow errors can occur.

Queue:

- Only front and rear elements are directly accessible.
- Requires extra logic for circular implementation (in arrays).
- Cannot perform random access efficiently.

Real-World Use Cases

Stack:

- Undo/Redo in text editors
- Function call management in recursion
- Backtracking algorithms (like maze solving)
- Expression evaluation (infix to postfix)

Queue:

- CPU scheduling and process management
- Data buffering (e.g., IO streams, printers)
- Breadth-First Search (BFS) in graphs
- Customer service and ticketing systems

Stack vs Queue (Quick Comparison Table)

Feature	Stack	Queue
Access Order	LIFO (Last In, First Out)	FIFO (First In, First Out)
Insertion	Top	Rear
Deletion	Top	Front
Access	Only top element	Front and rear elements
Use Case	Backtracking, recursion	Scheduling, buffering
STL Header	`<stack>`	`<queue>`