Stack and Its Operations in C

Definition

A **Stack** is a linear data structure that follows the **LIFO (Last In First Out)** principle.
This means the last element inserted is the first one to be removed.

Basic Operations

Operation	Description
`push()`	Add an element to the top of stack
`pop()`	Remove and return the top element
`peek()`	Return the top element (without removing)
`isEmpty()`	Check if the stack is empty

C Code Implementation

```c
#include <stdio.h>

#define SIZE 100  // Maximum size of the stack

int stack[SIZE]; // Array to store stack elements
int top = -1;    // Initialize top to -1, indicating the stack is empty

// Push operation: Adds an element to the top of the stack
void push(int x) {
    if (top == SIZE - 1) {
        // Stack is full, can't push more elements
        printf("Stack Overflow\n");
    } else {
        // Increment top and insert the element
        stack[++top] = x;
        printf("%d pushed to stack.\n", x);
    }
}

// Pop operation: Removes and returns the top element from the stack
int pop() {
    if (top == -1) {
        // Stack is empty, nothing to pop
        printf("Stack Underflow\n");
        return -1; // Return a default value
    } else {
        // Return the top element and decrement top
```

```
            return stack[top--];
        }
    }

    // Peek operation: Returns the top element without removing it
    int peek() {
        if (top == -1) {
            printf("Stack is Empty\n");
            return -1;
        } else {
            return stack[top];
        }
    }

    // isEmpty operation: Checks if the stack is empty
    int isEmpty() {
        return top == -1;
    }

    // Driver Code to demonstrate stack operations
    int main() {
        push(10);      // Push 10
        push(20);      // Push 20
        push(30);      // Push 30

        printf("Top element is %d\n", peek()); // Should print 30

        printf("Popped element is %d\n", pop()); // Should remove 30
        printf("Popped element is %d\n", pop()); // Should remove 20

        if (isEmpty())
            printf("Stack is empty\n");
        else
            printf("Stack is not empty\n");

        return 0;
    }


/*
    Output:
    ------------------------
    10 pushed to stack.
    20 pushed to stack.
    30 pushed to stack.
    Top element is 30
    Popped element is 30
    Popped element is 20
    Stack is not empty
*/
```

Queue – Data Structure Overview

What is a Queue?

A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle.

The element that is added **first** will be removed **first**.

Basic Queue Operations

Function	Purpose
`enqueue(x)`	Adds element `x` to the **rear** of the queue (using circular increment if needed)
`dequeue()`	Removes and returns the **front** element of the queue
`peek()`	Returns the front element **without removing** it
`isEmpty()`	Checks if the circular queue is **empty** (returns `1` or `0`)

Code :

```c
#include <stdio.h>

#define SIZE 100  // Maximum size of the queue

int queue[SIZE]; // Array to store queue elements
int front = -1;  // Index of the front element
int rear = -1;   // Index of the rear element

// Enqueue operation: Adds an element to the rear of the queue
void enqueue(int x) {
    if (rear == SIZE - 1) {
        // Queue is full
        printf("Queue Overflow\n");
    } else {
        if (front == -1) front = 0; // Set front to 0 on first insertion
        queue[++rear] = x;          // Insert element and move rear
        printf("%d enqueued to queue.\n", x);
    }
}

// Dequeue operation: Removes and returns the front element of the queue
int dequeue() {
    if (front == -1 || front > rear) {
        // Queue is empty
        printf("Queue Underflow\n");
        return -1;
    } else {
```

```
        // Return the front element and move front
        return queue[front++];
    }
}

// Peek operation: Returns the front element without removing it
int peek() {
    if (front == -1 || front > rear) {
        printf("Queue is Empty\n");
        return -1;
    } else {
        return queue[front];
    }
}

// isEmpty operation: Checks if the queue is empty
int isEmpty() {
    return front == -1 || front > rear;
}

// Driver Code to demonstrate queue operations
int main() {
    enqueue(10);      // Add 10
    enqueue(20);      // Add 20
    enqueue(30);      // Add 30

    printf("Front element is %d\n", peek()); // Should print 10

    printf("Dequeued element is %d\n", dequeue()); // Should remove 10
    printf("Dequeued element is %d\n", dequeue()); // Should remove 20

    if (isEmpty())
        printf("Queue is empty\n");
    else
        printf("Queue is not empty\n");

    return 0;
}

/*
    Output:
    -------------------------------
    10 enqueued to queue.
    20 enqueued to queue.
    30 enqueued to queue.
    Front element is 10
    Dequeued element is 10
    Dequeued element is 20
    Queue is not empty
*/
```

Infix to Postfix Conversion in C

What is Infix to Postfix Conversion?

- **Infix Expression**: Operators are placed **between operands**
  - Example: `A + B * C`
- **Postfix Expression**: Operators come **after operands**
  - Example: `A B C * +`

Why Convert to Postfix?

Postfix expressions are:

- Easier for **computers to evaluate**
- Don't require parentheses
- Ideal for **stack-based evaluation**

Operator Precedence and Associativity

Operator	Precedence	Associativity
`* / %`	High	Left to Right
`+ -`	Low	Left to Right

Example Conversion

Input (Infix): `A + B * C`
Output (Postfix): `A B C * +`
Reason: `*` has **higher precedence** than `+`

C Program: Infix to Postfix Conversion

```c
#include <stdio.h>
#include <ctype.h>   // for isalpha() and isdigit()
#include <string.h>  // for strlen()

#define SIZE 100

char stack[SIZE];
int top = -1;

// Function to push element onto the stack
```

```c
void push(char ch) {
    stack[++top] = ch;
}

// Function to pop element from the stack
char pop() {
    return stack[top--];
}

// Function to get the top element of the stack
char peek() {
    return stack[top];
}

// Function to check if the stack is empty
int isEmpty() {
    return top == -1;
}

// Function to return precedence of operators
int precedence(char op) {
    if (op == '^') return 3;
    if (op == '*' || op == '/' || op == '%') return 2;
    if (op == '+' || op == '-') return 1;
    return 0;
}

// Main function to convert infix to postfix
void infixToPostfix(char infix[]) {
    char postfix[SIZE];
    int i, j = 0;
    char ch;

    for (i = 0; infix[i] != '\0'; i++) {
        ch = infix[i];

        // If operand, add to postfix output
        if (isalnum(ch)) {
            postfix[j++] = ch;
        }
        // If '(', push to stack
        else if (ch == '(') {
            push(ch);
        }
        // If ')', pop until '('
        else if (ch == ')') {
            while (!isEmpty() && peek() != '(') {
                postfix[j++] = pop();
            }
            pop(); // remove '(' from stack
        }
        // If operator
        else {
```

```
            while (!isEmpty() && precedence(peek()) >= precedence(ch)) {
                postfix[j++] = pop();
            }
            push(ch);
        }
    }

    // Pop remaining operators
    while (!isEmpty()) {
        postfix[j++] = pop();
    }

    postfix[j] = '\0'; // null terminate the result

    printf("Postfix Expression: %s\n", postfix);
}

// Driver Code
int main() {
    char infix[SIZE];

    printf("Enter Infix Expression: ");
    scanf("%s", infix);

    infixToPostfix(infix);

    return 0;
}

    /*
        Output:
        ------------------------
        Enter Infix Expression: A+B*C
        Postfix Expression: ABC*+

    */
```

Postfix Expression Evaluation in C

What is Postfix Expression Evaluation?

In **Postfix Notation** (also known as **Reverse Polish Notation**):

- Operators come **after operands**
- Evaluated using a **stack** data structure

How it works:

1. **Operands** are **pushed** onto the stack

2. When an **operator** is encountered:
    - **Pop** the top two operands
    - **Apply** the operator
    - **Push** the result back to the stack

Example

Postfix Expression:

5 3 2 * +

Result:

11

Code:

```c
#include <stdio.h>
#include <ctype.h>   // for isdigit()
#include <stdlib.h>  // for atoi()
#include <string.h>  // for strlen()

#define SIZE 100

int stack[SIZE];
int top = -1;

// Function to push an element onto the stack
void push(int val) {
    if (top >= SIZE - 1) {
        printf("Stack Overflow\n");
        exit(1);
    }
    stack[++top] = val;
}

// Function to pop an element from the stack
int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        exit(1);
    }
    return stack[top--];
}

// Function to evaluate postfix expression
int evaluatePostfix(char expr[]) {
    int i;
    char ch;
```

```
    for (i = 0; expr[i] != '\0'; i++) {
        ch = expr[i];

        // Skip spaces
        if (ch == ' ' || ch == '\n')
            continue;

        // If digit, convert to number and push to stack
        if (isdigit(ch)) {
            int num = 0;
            while (isdigit(expr[i])) {
                num = num * 10 + (expr[i] - '0');
                i++;
            }
            i--; // Adjust i after overshooting
            push(num);
        }
        // If operator, pop two elements and apply operation
        else {
            int val2 = pop();
            int val1 = pop();
            int result;

            switch (ch) {
                case '+': result = val1 + val2; break;
                case '-': result = val1 - val2; break;
                case '*': result = val1 * val2; break;
                case '/':
                    if (val2 == 0) {
                        printf("Division by zero error!\n");
                        exit(1);
                    }
                    result = val1 / val2;
                    break;
                case '%':
                    if (val2 == 0) {
                        printf("Modulo by zero error!\n");
                        exit(1);
                    }
                    result = val1 % val2;
                    break;
                default:
                    printf("Invalid operator: %c\n", ch);
                    exit(1);
            }

            push(result);
        }
    }

    // Final result is on top
    return pop();
```

```c
}

// Driver Code
int main() {
    char postfixExpr[SIZE];

    printf("Enter Postfix Expression (space-separated, e.g., 5 3 2 * +):\n");
    fgets(postfixExpr, SIZE, stdin);

    // Remove newline if present
    size_t len = strlen(postfixExpr);
    if (postfixExpr[len - 1] == '\n') {
        postfixExpr[len - 1] = '\0';
    }

    int result = evaluatePostfix(postfixExpr);
    printf("Evaluated Result: %d\n", result);

    return 0;
}

/*
    Output:
    ----------------------------
    Enter Postfix Expression (space-separated, e.g., 5 3 2 * +):
    5 3 2 * +
    Evaluated Result: 11

*/
```