Practicle 1:

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>


#define MAX 100 // Maximum capacity of the stack


typedef struct {

    int data[MAX];

    int top;

} Stack;


// Initialize the stack

void initStack(Stack *s) {

    s->top = -1;

}


// Check if the stack is empty

int isEmpty(Stack *s) {

    return s->top == -1;

}


// Check if the stack is full

int isFull(Stack *s) {

    return s->top == MAX - 1;

}


// Push an element onto the stack

void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow!\n");
        return;
    }
    s->data[++(s->top)] = value;
}

// Pop an element from the stack
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow!\n");
        return INT_MIN;
    }
    return s->data[(s->top)--];
}

// Peek the top element of the stack
int top(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return INT_MIN;
    }
    return s->data[s->top];
}

// Display all elements of the stack
void display(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = 0; i <= s->top; i++) {
        printf("%d ", s->data[i]);
    }
    printf("\n");
}

// MinStack Implementation
typedef struct {
    Stack mainStack;
    Stack minStack;
} MinStack;

// Initialize MinStack
void initMinStack(MinStack *ms) {
    initStack(&ms->mainStack);
    initStack(&ms->minStack);
}

// Push an element onto the MinStack
void minStackPush(MinStack *ms, int value) {
    push(&ms->mainStack, value);
    if (isEmpty(&ms->minStack) || value <= top(&ms->minStack)) {
        push(&ms->minStack, value);
    }
}
```

```
// Pop an element from the MinStack
int minStackPop(MinStack *ms) {
   if (isEmpty(&ms->mainStack)) {
      printf("Stack Underflow!\n");
      return INT_MIN;
   }
   int popped = pop(&ms->mainStack);
   if (popped == top(&ms->minStack)) {
      pop(&ms->minStack);
   }
   return popped;
}

// Retrieve the minimum element from the MinStack
int getMin(MinStack *ms) {
   if (isEmpty(&ms->minStack)) {
      printf("Stack is empty!\n");
      return INT_MIN;
   }
   return top(&ms->minStack);
}

// Display MinStack
void displayMinStack(MinStack *ms) {
   printf("Main Stack: ");
   display(&ms->mainStack);
   printf("Min Stack: ");
   display(&ms->minStack);
}
```

```
// Main function to test MinStack
int main() {
    MinStack ms;
    initMinStack(&ms);

    int choice, value;

    do {
        printf("\nMenu:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Top Element\n");
        printf("4. Get Minimum\n");
        printf("5. Display Stack\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value to push: ");
                scanf("%d", &value);
                minStackPush(&ms, value);
                break;

            case 2:
                value = minStackPop(&ms);
                if (value != INT_MIN)
                    printf("Popped element: %d\n", value);
                break;

            case 3:
                value = top(&ms.mainStack);
                if (value != INT_MIN)
                    printf("Top element: %d\n", value);
                break;

            case 4:
                value = getMin(&ms);
                if (value != INT_MIN)
                    printf("Minimum element: %d\n", value);
                break;

            case 5:
                displayMinStack(&ms);
                break;

            case 6:
                printf("Exiting...\n");
                break;

            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 6);

    return 0;
```

```
}
```

OUTPUT:

```
Menu:

1. Push

2. Pop

3. Top Element

4. Get Minimum

5. Display Stack

6. Exit

Enter your choice:
```

2.Write a program to deal with real-world situations where Stack data structure is widely used

Evaluation of expression: Stacks are used to evaluate expressions, especially in languages that use postfix or prefix notation. Operators and operands are pushed onto the stack, and operations are performed based on the LIFO principle.

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>


#define MAX 100  // Maximum stack size


// Define the stack structure

struct Stack {

   int arr[MAX];

   int top;

};
```

```
// Initialize the stack
void initialize(struct Stack *stack) {
    stack->top = -1;  // Indicates the stack is empty
}

// Check if the stack is empty
int isEmpty(struct Stack *stack) {
    return stack->top == -1;
}

// Push operation
void push(struct Stack *stack, int value) {
    if (stack->top == MAX - 1) {
        printf("Stack overflow! Cannot push %d.\n", value);
    } else {
        stack->arr[++stack->top] = value;
    }
}

// Pop operation
int pop(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow! Cannot pop.\n");
        return -1;
    } else {
        return stack->arr[stack->top--];
    }
}
```

```
// Function to evaluate a postfix expression
int evaluatePostfix(char* expression) {
    struct Stack s;
    initialize(&s);

    for (int i = 0; expression[i] != '\0'; i++) {
        if (isdigit(expression[i])) {
            // If the character is a digit, push it onto the stack
            push(&s, expression[i] - '0');  // Convert character to integer
        } else {
            // If the character is an operator, pop two operands and perform the operation
            int operand2 = pop(&s);
            int operand1 = pop(&s);
            switch (expression[i]) {
                case '+': push(&s, operand1 + operand2); break;
                case '-': push(&s, operand1 - operand2); break;
                case '*': push(&s, operand1 * operand2); break;
                case '/': push(&s, operand1 / operand2); break;
                default:
                    printf("Invalid operator encountered: %c\n", expression[i]);
                    return -1;
            }
        }
    }
    return pop(&s);  // The final result will be the last remaining element in the stack
}

// Main function
int main() {
    char expression[MAX];

    // Input the postfix expression
    printf("Enter a postfix expression: ");
    scanf("%s", expression);

    int result = evaluatePostfix(expression);
    printf("Result of the postfix expression: %d\n", result);

    return 0;
}
```

Input: 53+82-*

RESULT : Result of the postfix expression: 48

3. Write a program for finding NGE NEXT GREATER ELEMENT from an array

```
#include <stdio.h>
#define MAX 100

// Function to find the Next Greater Element
void findNextGreaterElement(int arr[], int n) {
    int stack[MAX];  // Stack to store elements
    int top = -1;    // Initialize stack as empty
    int nge[MAX];    // Array to store NGE for each element

    // Traverse the array from right to left
    for (int i = n - 1; i >= 0; i--) {
        // Remove elements smaller than or equal to the current element
        while (top != -1 && stack[top] <= arr[i]) {
            top--;
        }

        // If stack is not empty, the top element is the NGE
        if (top != -1) {
            nge[i] = stack[top];
        } else {
            nge[i] = -1;  // No greater element found
        }

        // Push the current element onto the stack
        stack[++top] = arr[i];
    }

    // Print the results
    printf("Next Greater Elements:\n");
    for (int i = 0; i < n; i++) {
        printf("%d -> %d\n", arr[i], nge[i]);
    }
}

// Main function
int main() {
    int arr[MAX];
    int n;

    // Input the array size and elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    printf("Enter the array elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Find and print NGE for each element
    findNextGreaterElement(arr, n);

    return 0;
}
```

```
INPUT: Enter the number of elements: 6
Enter the array elements: 4 5 2 25 7 8

OUTPUT: Next Greater Elements:
4 -> 5
5 -> 25
2 -> 25
25 -> -1
7 -> 8
8 -> -1
```

4. Write a program to design a circular queue(k) which Should implement the below functions

a. Enqueue

b. Dequeue

c. Front

d. Rear

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 5  // Define the maximum size of the circular queue

// Define the Circular Queue structure
struct CircularQueue {
    int arr[MAX];
    int front;
    int rear;
};

// Initialize the queue
void initialize(struct CircularQueue *q) {
    q->front = -1;
    q->rear = -1;
}

// Check if the queue is full
int isFull(struct CircularQueue *q) {
    return (q->front == 0 && q->rear == MAX - 1) || (q->rear == (q->front - 1) % (MAX - 1));
}

// Check if the queue is empty
int isEmpty(struct CircularQueue *q) {
    return q->front == -1;
}
```

```
// Enqueue operation
void enqueue(struct CircularQueue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue %d.\n", value);
        return;
    }
    if (isEmpty(q)) {  // First element insertion
        q->front = q->rear = 0;
    } else if (q->rear == MAX - 1 && q->front != 0) {  // Wrap around
        q->rear = 0;
    } else {
        q->rear++;
    }
    q->arr[q->rear] = value;
    printf("%d enqueued to the queue.\n", value);
}

// Dequeue operation
int dequeue(struct CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }
    int data = q->arr[q->front];
    if (q->front == q->rear) {  // Only one element was present
        q->front = q->rear = -1;
    } else if (q->front == MAX - 1) {  // Wrap around
        q->front = 0;
    } else {
        q->front++;
    }
    return data;
}

// Get the front element
int front(struct CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No front element.\n");
        return -1;
    }
    return q->arr[q->front];
}

// Get the rear element
int rear(struct CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No rear element.\n");
        return -1;
    }
    return q->arr[q->rear];
}

// Main function to demonstrate the circular queue
int main() {
    struct CircularQueue q;
    initialize(&q);
    // Test the circular queue operations
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);
    enqueue(&q, 50);  // Queue is now full

    printf("Front element: %d\n", front(&q));
    printf("Rear element: %d\n", rear(&q));

    printf("Dequeued: %d\n", dequeue(&q));
    enqueue(&q, 60);  // Demonstrates wrap-around

    printf("Front element: %d\n", front(&q));
    printf("Rear element: %d\n", rear(&q));

    return 0;
}
```

```
OUTPUT: 10 enqueued to the queue.
20 enqueued to the queue.
30 enqueued to the queue.
40 enqueued to the queue.
50 enqueued to the queue.
Front element: 10
Rear element: 50
Dequeued: 10
60 enqueued to the queue.
Front element: 20
```

```
Rear element: 60
```

5. Write a Program for an infix expression, and convert it to postfix notation. Use a queue to implement the Shunting Yard Algorithm for expression conversion.

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>


#define MAX 100


// Define the queue structure

typedef struct {

   char arr[MAX][MAX];  // Queue to store strings (operands/operators)

   int front, rear;

} Queue;


// Define the stack structure

typedef struct {

   char arr[MAX];

   int top;

} Stack;


// Queue operations

void initQueue(Queue* q) {

   q->front = q->rear = -1;

}
```

```
int isQueueEmpty(Queue* q) {
    return q->front == -1;
}

void enqueue(Queue* q, char* value) {
    if (q->rear == MAX - 1) {
        printf("Queue overflow!\n");
        return;
    }
    if (q->front == -1) q->front = 0;
    strcpy(q->arr[++q->rear], value);
}

void displayQueue(Queue* q) {
    if (isQueueEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    for (int i = q->front; i <= q->rear; i++) {
        printf("%s ", q->arr[i]);
    }
    printf("\n");
}

// Stack operations
void initStack(Stack* s) {
    s->top = -1;
}
```

```
int isStackEmpty(Stack* s) {
    return s->top == -1;
}

char peek(Stack* s) {
    return s->arr[s->top];
}

void push(Stack* s, char value) {
    if (s->top == MAX - 1) {
        printf("Stack overflow!\n");
        return;
    }
    s->arr[++s->top] = value;
}

char pop(Stack* s) {
    if (isStackEmpty(s)) {
        printf("Stack underflow!\n");
        return '\0';
    }
    return s->arr[s->top--];
}

// Helper functions
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
```

```
}

int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

// Shunting Yard Algorithm
void infixToPostfix(char* infix) {
    Stack s;
    Queue q;
    initStack(&s);
    initQueue(&q);

    char token[MAX];
    int i = 0;

    while (infix[i] != '\0') {
        if (isdigit(infix[i])) {
            // If token is an operand, enqueue it
            int j = 0;
            while (isdigit(infix[i])) {
                token[j++] = infix[i++];
            }
            token[j] = '\0';
            enqueue(&q, token);
        } else if (infix[i] == '(') {
            // Push '(' to the stack
            push(&s, infix[i]);
            i++;
        } else if (infix[i] == ')') {
            // Pop until '(' is encountered
            while (!isStackEmpty(&s) && peek(&s) != '(') {
                char op[2] = {pop(&s), '\0'};
                enqueue(&q, op);
            }
            pop(&s); // Remove '('
            i++;
        } else if (isOperator(infix[i])) {
            // Handle operators
            while (!isStackEmpty(&s) && precedence(peek(&s)) >= precedence(infix[i])) {
                char op[2] = {pop(&s), '\0'};
                enqueue(&q, op);
            }
            push(&s, infix[i]);
            i++;
        } else {
            i++; // Ignore spaces
        }
    }

    // Pop any remaining operators
    while (!isStackEmpty(&s)) {
        char op[2] = {pop(&s), '\0'};
        enqueue(&q, op);
    }

    // Display the postfix expression
    printf("Postfix Expression: ");
```

```
    displayQueue(&q);
}


// Main function
int main() {
    char infix[MAX];
    printf("Enter an infix expression: ");
    fgets(infix, MAX, stdin);  // Read the input


    infix[strcspn(infix, "\n")] = '\0';  // Remove newline character


    infixToPostfix(infix);


    return 0;
}
```

```
INPUT: Enter an infix expression: 3 + 5 * 2 - 8 / 4
OUTPUT: Postfix Expression: 3 5 2 * + 8 4 / -
```

SHUTTING YARD ALGO:

In computer science, the shunting-yard algorithm is a method for parsing mathematical expressions specified in infix notation. It can be used to produce output in Reverse Polish notation or as an abstract syntax tree. The algorithm was invented by Edsger Dijkstra and named the "shunting yard" algorithm because its operation resembles that of a railroad shunting yard. Dijkstra first described the Shunting Yard Algorithm in the Mathematisch Centrum report MR 34/61. Like the evaluation of RPN, the shunting yard algorithm is stack-based. Infix expressions are the form of mathematical notation most people are used to, for instance 3+4 or 3+4*(2−1). For the conversion there are two text variables, the input and the output. There is also a stack that holds operators not yet added to the output queue. To convert, the program reads each symbol in order and does something based on that symbol.

Difference between Shunting Yard Algorithm and Infix-to-Postfix Conversion:

The Shunting Yard Algorithm **implements** the logic to convert infix to postfix notation. So, converting an infix expression to postfix typically means using the Shunting Yard Algorithm

How the Shunting Yard Algorithm Works with a Stack:

- **Stack (LIFO)** ensures that the most recent operator is processed first, preserving operator precedence and ensuring correct order for postfix conversion.
- Operators are pushed onto the stack and popped when needed to maintain the correct order of operations.

What Happens if We Use a Queue for Operators:

A **queue (FIFO)** processes items in the order they were added. This disrupts the operator precedence and associativity rules essential for correct postfix conversion.

Consequences of Using a Queue:

1. **Loss of Precedence Control:**
   - Higher-precedence operators (e.g., *, /) might be enqueued and processed **after** lower-precedence operators (e.g., +, -), leading to incorrect order.
2. **Incorrect Associativity Handling:**
   - For operators with the same precedence (like + and -), left-to-right associativity requires handling the most recent operator first. A queue processes the oldest operator first, violating this rule.
3. **Output Errors:**
   - The final postfix expression may not reflect the correct order of operations, leading to incorrect results when evaluated.

Example Scenario:

Consider the infix expression:
3 + 4 * 2

- **Correct Postfix (using a stack):** 3 4 2 * +
- **Incorrect Postfix (if a queue is used for operators):** 3 + 4 2 *

Why This Happens:

- **Stacks** ensure that operators with higher precedence stay on top, delaying the processing of lower-precedence operators until the stack is clear.
- A **queue** would process operators in the order they appear, regardless of precedence, leading to flawed postfix notation.

Conclusion:

To properly convert an infix expression to postfix notation using the Shunting Yard Algorithm:

- **Operators must be managed using a stack** because it preserves precedence and associativity rules.
- **Queues** are appropriate for storing the output expression but not for handling operators.

Using a queue for operators fundamentally breaks the algorithm's logic, producing incorrect results.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Define structure for Queue
typedef struct {
  char items[100][10]; // Store tokens (operators/operands)
  int front;
  int rear;
} Queue;

// Initialize a queue
void initQueue(Queue *q) {
  q->front = -1;
  q->rear = -1;
}
```

```
// Check if the queue is empty
int isEmpty(Queue *q) {
    return q->rear == -1;
}

// Enqueue an item into the queue
void enqueue(Queue *q, char *value) {
    if (q->rear == 99) return; // Full queue
    if (q->front == -1) q->front = 0;
    q->rear++;
    strcpy(q->items[q->rear], value);
}

// Dequeue an item from the queue
char *dequeue(Queue *q) {
    if (isEmpty(q)) return NULL;
    char *item = q->items[q->front];
    if (q->front >= q->rear) { // Only one element left
        q->front = q->rear = -1;
    } else {
        q->front++;
    }
    return item;
}

// Function to return precedence of operators
int precedence(char *op) {
    if (strcmp(op, "+") == 0 || strcmp(op, "-") == 0) return 1;
    if (strcmp(op, "*") == 0 || strcmp(op, "/") == 0) return 2;
    return 0;
}


// Check if the token is an operator
int isOperator(char *token) {
    return (strcmp(token, "+") == 0 || strcmp(token, "-") == 0 ||
            strcmp(token, "*") == 0 || strcmp(token, "/") == 0);
}


// Convert infix to postfix using the Shunting Yard Algorithm with a Queue (incorrect logic)
void infixToPostfix(char *infix) {
    char postfix[100][10]; // Array to store the postfix expression
    int postfixIndex = 0;


    Queue operatorQueue;
    initQueue(&operatorQueue);


    char *token = strtok(infix, " "); // Tokenize the infix expression
    while (token != NULL) {
        if (isdigit(token[0])) { // If operand, add to postfix
            strcpy(postfix[postfixIndex++], token);
        } else if (isOperator(token)) {
            // Process operators (incorrect logic with queue)
            while (!isEmpty(&operatorQueue) && precedence(operatorQueue.items[operatorQueue.front])
>= precedence(token)) {
                strcpy(postfix[postfixIndex++], dequeue(&operatorQueue));
            }
            enqueue(&operatorQueue, token);
        }
        token = strtok(NULL, " ");
    }

    // Dequeue any remaining operators
    while (!isEmpty(&operatorQueue)) {
        strcpy(postfix[postfixIndex++], dequeue(&operatorQueue));
    }

    // Print postfix expression
    printf("Postfix Expression: ");
    for (int i = 0; i < postfixIndex; i++) {
        printf("%s ", postfix[i]);
    }
    printf("\n");
}

// Example usage
int main() {
    char infixExpression[] = "3 + 4 * 2";
    printf("Infix Expression: %s\n", infixExpression);
    infixToPostfix(infixExpression);
    return 0;
}
```

Output:

```
 Infix Expression: 3 + 4 * 2
Postfix Expression: 3 + 4 2 *
```

Key Issues in the Code:

1. **Incorrect Operator Handling**:
   Operators are queued and processed in the order they appear, leading to incorrect precedence handling.
2. **Lack of Associativity Management**:
   Operators are processed in FIFO order rather than LIFO (Last-In, First-Out), which breaks the correct handling of associativity.

Why Use a Stack Instead of a Queue?

- **Precedence Handling**: Stacks ensure that higher-precedence operators stay on top and are processed first.
- **Correct Order of Operations**: With a stack, you can manage left-to-right associativity and ensure operators are popped at the correct time.

This code serves as a learning tool to understand why **stacks are essential** in the Shunting Yard Algorithm. Using a queue for operators disrupts the logic, producing incorrect results.