// PROBLEM STATEMENT:

// Write a C program to evaluate the given postfix expression using stack.

// CODE:

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

// Define maximum stack size
#define MAX 100

// Define Stack structure
struct Stack {
   int top;
   int array[MAX];
};

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

// Function to pop an element from the stack
char pop(struct Stack * stack) {
   if (!isEmpty(stack)) return stack -> array[stack -> top--];
}

// Function to push an element onto the stack
void push(struct Stack * stack, char op) {
   stack -> array[++stack -> top] = op;
}

// Function to evaluate a postfix expression
int postfix(char * exp) {
   // Allocate memory for stack and initialize top to -1
   struct Stack * stack = (struct Stack * ) malloc(sizeof(struct Stack));
   stack -> top = -1;

   int i;

   // Check if memory allocation is successful
   if (!stack) return -1;

   // Loop through each character in the expression
   for (i = 0; exp[i]; ++i) {
      // If the current character is a digit, push it onto the stack
```

```
        if (isdigit(exp[i])) push(stack, exp[i] - '0');

        // If the current character is an operator, apply the operator to the top
        // two elements on the stack
        else {
            int val1 = pop(stack);
            int val2 = pop(stack);

            // Perform the corresponding operation and push the result back onto the
            // stack
            switch (exp[i]) {
            case '+':
                push(stack, val2 + val1);
                break;
            case '-':
                push(stack, val2 - val1);
                break;
            case '*':
                push(stack, val2 * val1);
                break;
            case '/':
                push(stack, val2 / val1);
                break;
            case '%':
                push(stack, val2 % val1);
                break;
            case '^':
                push(stack, pow(val2, val1));
                break;
            }
        }
    }
    // Return the top element of the stack, which is the final result of the
    // evaluation
    return pop(stack);
}

// Main function
int main() {
    // Define a postfix expression
    char exp[100];
    printf("Enter postfix expression:\n");
    gets(exp);

    // Evaluate the postfix expression and print the result
    printf("Postfix evaluation: %d", postfix(exp));
    return 0;
}
```

```
// OUTPUT:

Enter postfix expression:
95%2*
Postfix evaluation: 8

Enter postfix expression:
23^52*-
Postfix evaluation: -2
```

AYUSH RAWAT MCA-B 25 Std. ID - 22391138

```
// PROBLEM STATEMENT:
// Write a C program to convert a given infix expression into the equivalent postfix
// expression using stack array.

// CODE:

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

#define MAX_SIZE 100

// Function to check if a character is an operator
int is_operator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^' ||
        ch == '%';
}

// Function to determine the precedence of an operator
int precedence(char ch) {
    switch (ch) {
    case '^':
        return 3;
    case '*':
    case '/':
    case '%':
        return 2;
    case '+':
    case '-':
        return 1;
    default:
        return 0;
    }
}

// Function to push a character onto the stack
void push(char stack[], int * top, char ch) {
    if ( * top >= MAX_SIZE - 1) {
        printf("Stack overflow\n");
        exit(1);
    }
    stack[++( * top)] = ch;
}

// Function to pop a character from the stack
char pop(char stack[], int * top) {
    if ( * top < 0) {
        printf("Stack underflow\n");
        exit(1);
    }
    return stack[( * top) --];
}

// Function to peek at the top character of the stack
char peek(char stack[], int top) {
    return stack[top];
}

void postfix(char infix[], int n) {
    char postfix[MAX_SIZE], stack[MAX_SIZE];
    int i, j, top = -1;

    // Convert infix expression to postfix expression
    for (i = 0, j = 0; i < n - 1; i++) {
        if (isalnum(infix[i])) {
            postfix[j++] = infix[i];
        } else if (infix[i] == '(') {
            push(stack, & top, infix[i]);
        } else if (infix[i] == ')') {
            while (peek(stack, top) != '(') {
                postfix[j++] = pop(stack, & top);
            }
            pop(stack, & top); // Remove '(' from stack
        } else if (is_operator(infix[i])) {
            while (top != -1 &&
                precedence(peek(stack, top)) >= precedence(infix[i])) {
                postfix[j++] = pop(stack, & top);
            }
            push(stack, & top, infix[i]);
        }
    }

    // Pop any remaining operators from stack
    while (top != -1) {
        postfix[j++] = pop(stack, & top);
    }

    // Add null-terminator to postfix string
    postfix[j] = '\0';

    // Print postfix expression
    printf("Postfix expression: %s\n", postfix);
}

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

    postfix(infix, strlen(infix));
    return 0;
}


// OUTPUT:

Enter infix expression: 2*3+4-3+6*3/5
Postfix expression: 23*4+3-63*5/+


Enter infix expression: 7*8/2+4-9^3
Postfix expression: 78*2/4+93^-
```

```
// PROBLEM STATEMENT:
// Write a program in C to add two polynomial equations using linked list

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

// Define the structure for a term in the polynomial
typedef struct Node {
    int coefficient;
    int exponent;
    struct Node * next;
} Node;

// Function to create a new node with given coefficient and exponent
Node * createNode(int coefficient, int exponent) {
    Node * newNode = (Node * ) malloc(sizeof(Node));
    newNode -> coefficient = coefficient;
    newNode -> exponent = exponent;
    newNode -> next = NULL;
    return newNode;
}

// Function to insert a node at the end of the linked list
void insert(Node ** head, int coefficient, int exponent) {
    Node * newNode = createNode(coefficient, exponent);
    if ( * head == NULL) {
        * head = newNode;
        return;
    }

    Node * temp = * head;
    while (temp -> next != NULL) {
        temp = temp -> next;
    }
    temp -> next = newNode;
}

// Function to add two polynomials using linked lists
Node * addPolynomials(Node * p1, Node * p2) {
    Node * result = NULL;

    while (p1 && p2) {
        if (p1 -> exponent > p2 -> exponent) {
            insert( & result, p1 -> coefficient, p1 -> exponent);
            p1 = p1 -> next;
        } else if (p1 -> exponent < p2 -> exponent) {
            insert( & result, p2 -> coefficient, p2 -> exponent);
            p2 = p2 -> next;
        } else {
            int sum = p1 -> coefficient + p2 -> coefficient;
            if (sum != 0) {
                insert( & result, sum, p1 -> exponent);
            }
            p1 = p1 -> next;
            p2 = p2 -> next;
        }
    }

    // Add remaining terms from p1
    while (p1) {
        insert( & result, p1 -> coefficient, p1 -> exponent);
        p1 = p1 -> next;
    }

    // Add remaining terms from p2
    while (p2) {
        insert( & result, p2 -> coefficient, p2 -> exponent);
        p2 = p2 -> next;
    }

    return result;
}

// Function to display the polynomial
void display(Node * head) {
    if (head == NULL) {
        printf("Empty polynomial");
        return;
    }

    Node * temp = head;
    while (temp != NULL) {
        printf("%dx^%d", temp -> coefficient, temp -> exponent);
        if (temp -> next != NULL) {
            printf(" + ");
        }
        temp = temp -> next;
    }
    printf("\n");
}

int main() {
    Node * p1 = NULL, * p2 = NULL, * result = NULL;

    // Insert terms for the first polynomial
    insert( & p1, 8, 2);
    insert( & p1, 3, 1);
    insert( & p1, 4, 0);

    // Insert terms for the second polynomial
    insert( & p2, 3, 3);
    insert( & p2, 4, 2);
    insert( & p2, 1, 1);
    insert( & p2, 5, 0);

    // Display the polynomials
    printf("Polynomial 1: ");
    display(p1);
    printf("Polynomial 2: ");
    display(p2);

    // Add the polynomials and display the result
    result = addPolynomials(p1, p2);
    printf("Result: ");
    display(result);

    return 0;
}
```

```
// OUTPUT:

Polynomial 1: 8x^2 + 3x^1 + 4x^0
Polynomial 2: 3x^3 + 4x^2 + 1x^1 + 5x^0
Result: 3x^3 + 12x^2 + 4x^1 + 9x^0
```