1.main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include "header.h"
#include <string.h>

int main(){
    char infix[100], postfix[100], prefix[100];
    printf("Enter a valid infix expression: ");
    scanf("%[^\n]s", infix);

    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);

    int result = evaluatePostfix(postfix);

    // Resetting input string for infix to prefix conversion
    // strcpy(prefix, infix); // Storing the original infix expression before conversion
    infixToPrefix(infix, prefix);
    printf("Prefix expression: %s\n", prefix);

    printf("Evaluated result: %d\n", result);
    return 0;
}
```

2.header.h

```
typedef struct {
   int top;
   int arr[100];
} Stack;

void init(Stack *s);
int isFull(Stack *s);
int isEmpty(Stack *s);
void push(Stack *s, int value);
int pop(Stack *s);
int peek(Stack *s);

void infixToPostfix(char* infix, char* postfix);
int evaluatePostfix(char* postfix);

void reverseString(char* str);
void replaceParentheses(char* expr);
void infixToPrefix(char* infix, char* prefix);
```

3.logic.c

```
#include "header.h"
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <string.h>

void init(Stack *s) {
    s->top = -1;
}

int isFull(Stack *s) {
    return s->top == 99; // MAX size is 100
}

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

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

int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
```

```
        return 0;
    }
    return s->arr[s->top--];
}

int peek(Stack *s) {
    if (isEmpty(s)) {
        return 0;
    }
    return s->arr[s->top];
}

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

int isLeftAssociative(char op) {
    return op != '^'; // All operators except '^' are left associative
}

void infixToPostfix(char* infix, char* postfix) {
    Stack operators;
    init(&operators);

    int i = 0, k = 0;
    while (infix[i] != '\0') {
        if (isspace(infix[i])) {
            i++;
```

```
        continue;
    }
    if (isdigit(infix[i])) {
        while (isdigit(infix[i])) {
            postfix[k++] = infix[i++];
        }
        postfix[k++] = ' ';
        continue;
    }

    if (infix[i] == '(') {
        push(&operators, infix[i]);
    } else if (infix[i] == ')') {
        while (!isEmpty(&operators) && peek(&operators) != '(') {
            postfix[k++] = pop(&operators);
            postfix[k++] = ' ';
        }
        pop(&operators); // Remove '(' from the stack
    } else {
        while (!isEmpty(&operators) &&
               (precedence(peek(&operators)) > precedence(infix[i]) ||
               (precedence(peek(&operators)) == precedence(infix[i]) && isLeftAssociative(infix[i])))) {
            postfix[k++] = pop(&operators);
            postfix[k++] = ' ';
        }
        push(&operators, infix[i]);
    }
    i++;
}

while (!isEmpty(&operators)) {
```

```
            postfix[k++] = pop(&operators);
            postfix[k++] = ' ';
        }
        postfix[k - 1] = '\0'; // Null terminate the postfix expression
    }
    void reverseString(char* str) {
        int length = strlen(str);
        for (int i = 0; i < length / 2; i++) {
            char temp = str[i];
            str[i] = str[length - i - 1];
            str[length - i - 1] = temp;
        }
    }
    void replaceParentheses(char* expr) {
        int i = 0;
        while (expr[i] != '\0') {
            if (expr[i] == '(') {
                expr[i] = ')';
            } else if (expr[i] == ')') {
                expr[i] = '(';
            }
            i++;
        }
    }
    void infixToPrefix(char* infix, char* prefix) {
        // Step 1: Reverse the infix expression
        reverseString(infix);
        // Step 2: Replace '(' with ')' and vice versa
        replaceParentheses(infix);
        // Step 3: Initialize a stack for operators
        Stack operators;
```

```
init(&operators);
int i = 0, k = 0;
char postfix[100];
while (infix[i] != '\0') {
   if (isspace(infix[i])) {
      i++;
      continue;
   }
   if (isdigit(infix[i])) {
      while (isdigit(infix[i])) {
         postfix[k++] = infix[i++];
      }
      postfix[k++] = ' ';
      continue;
   }
   if (infix[i] == '(') {
      push(&operators, infix[i]);
   } else if (infix[i] == ')') {
      while (!isEmpty(&operators) && peek(&operators) != '(') {
         postfix[k++] = pop(&operators);
         postfix[k++] = ' ';
      }
      pop(&operators); // Remove '(' from the stack
   } else { // Operator
      while (!isEmpty(&operators) &&
      precedence(peek(&operators)) > precedence(infix[i])) {
      postfix[k++] = pop(&operators);
      postfix[k++] = ' ';
      }
   // Always push the current operator onto the stack
   push(&operators, infix[i]);
```

```
        }
        i++;
    }
    while (!isEmpty(&operators)) {
        postfix[k++] = pop(&operators);
        postfix[k++] = ' ';
    }
    postfix[k - 1] = '\0'; // Null terminate the postfix expression
    // Step 4: Reverse the postfix expression to get the prefix expression
    reverseString(postfix);
    // Copy the final prefix expression to the prefix parameter
    strcpy(prefix, postfix);
}
int applyOperation(int a, int b, char op) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
        case '^': return (int)pow(a, b);
    }
    return 0;
}
int evaluatePostfix(char* postfix) {
    Stack operands;
    init(&operands);
    int i = 0;
    while (postfix[i] != '\0') {
        if (isspace(postfix[i])) {
            i++;
            continue;
```

```
        }
        if (isdigit(postfix[i])) {
            int num = 0;
            while (isdigit(postfix[i])) {
                num = num * 10 + (postfix[i] - '0');
                i++;
            }
            push(&operands, num);
        } else {
            int val2 = pop(&operands);
            int val1 = pop(&operands);
            int result = applyOperation(val1, val2, postfix[i]);
            push(&operands, result);
            i++;
        }
    }
    return pop(&operands);
}
```

Output:

```
tanis@Tanishq MINGW64 /d/COEP/DSA/Serious/LabWork-StackAndQueue/Stack Application
$ gcc -Wall main.c logic.c

tanis@Tanishq MINGW64 /d/COEP/DSA/Serious/LabWork-StackAndQueue/Stack Application
$ ./a
Enter a valid infix expression: (3 + 5) * (6 - 2) / 2
Postfix expression: 3 5 + 6 2 - * 2 /
Prefix expression: / * + 3 5 - 6 2 2
Evaluated result: 16

tanis@Tanishq MINGW64 /d/COEP/DSA/Serious/LabWork-StackAndQueue/Stack Application
$ ./a
Enter a valid infix expression: (2 + 3) * (4 - 1)
Postfix expression: 2 3 + 4 1 - *
Prefix expression: * + 2 3 - 4 1
Evaluated result: 15

tanis@Tanishq MINGW64 /d/COEP/DSA/Serious/LabWork-StackAndQueue/Stack Application
$ ./a
Enter a valid infix expression: 5 + 6 * 3 - 4 / 2
Postfix expression: 5 6 3 * + 4 2 / -
Prefix expression: - + 5 * 6 3 / 4 2
Evaluated result: 21
```

```
tanis@Tanishq MINGW64 /d/COEP/DSA/Serious/LabWork-StackAndQueue/Stack Application
$ ./a
Enter a valid infix expression: ((1 + 2) * (3 + 4)) / (5 - 1)
Postfix expression: 1 2 + 3 4 + * 5 1 - /
Prefix expression: / * + 1 2 + 3 4 - 5 1
Evaluated result: 5

tanis@Tanishq MINGW64 /d/COEP/DSA/Serious/LabWork-StackAndQueue/Stack Application
$ ./a
Enter a valid infix expression: 3 + 4 * 2 / (1 - 5) ^ 2 ^ 3
Postfix expression: 3 4 2 * 1 5 - 2 3 ^ ^ / +
Prefix expression: + 3 / * 4 2 ^ ^ - 1 5 2 3
Evaluated result: 3
```