Aim: Evaluate Postfix Expression using Stack ADT.

Theory:

A postfix expression is a mathematical notation in which the operator follows the operands (e.g., "a b +"). To evaluate a postfix expression, a stack-based approach is commonly used. The postfix evaluation process involves scanning the expression from left to right, pushing operands onto a stack, and applying operators to the two most recent operands on the stack whenever an operator is encountered. The result of each operation is pushed back onto the stack until the entire expression is evaluated. The final answer is the single remaining value in the stack.

Procedure:

1. Create an empty stack to store operands.
2. Traverse the postfix expression from left to right:
   - If the element is a number, push it onto the stack.
   - If the element is an operator, pop the top two operands from the stack, apply the operator, and push the result back onto the stack.
3. Repeat step 2 until the end of the expression.
4. The final result will be the last remaining element in the stack.

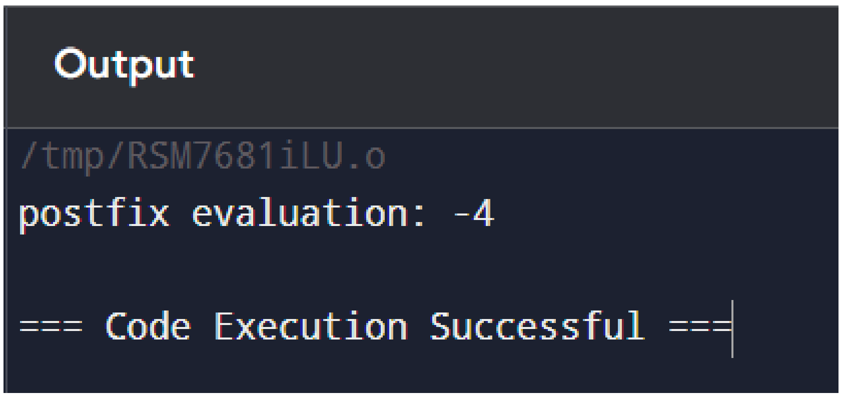
Code:

```
// C program to evaluate value of a postfix expression
 #include <ctype.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
 // Stack type
 struct Stack {
int top;
 unsigned capacity;
 int* array;
```

```
};
// Stack Operations
struct Stack* createStack(unsigned capacity)
{
struct Stack* stack
= (struct Stack*)malloc(sizeof(struct Stack));
if (!stack)
return NULL;
stack->top =-1;
stack->capacity = capacity;
stack->array
= (int*)malloc(stack->capacity * sizeof(int));
if (!stack->array)
return NULL;
return stack;
}
int isEmpty(struct Stack* stack)
{
return stack->top ==-1;
}
char peek(struct Stack* stack)
{
return stack->array[stack->top];
}
char pop(struct Stack* stack)
{
 if (!isEmpty(stack))
return stack->array[stack->top--];
 return '$';
 }
 void push(struct Stack* stack, char op)
 {
 stack->array[++stack->top] = op;
 }
 // The main function that returns value
 // of a given postfix expression
 int evaluatePostfix(char* exp)
 {
 // Create a stack of capacity equal to expression size
 struct Stack* stack = createStack(strlen(exp));
 int i;
 // See if stack was created successfully
 if (!stack)
 return-1;
 // Scan all characters one by one
 for (i = 0; exp[i]; ++i) {
 // If the scanned character is an operand
 // (number here), push it to the stack.
 if (isdigit(exp[i]))
 push(stack, exp[i]-'0');
 // If the scanned character is an operator,
 // pop two elements from stack apply the operator
 else {
 int val1 = pop(stack);
 int val2 = pop(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;
 }
 }
 }
 return pop(stack);
 }
 // Driver code
 int main()
 {
 char exp[] = "231*+9-";
 // Function call
```

```
printf("postfix evaluation: %d", evaluatePostfix(exp));
return 0;
}
```

Output:



```
/tmp/RSM7681iLU.o
postfix evaluation: -4

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

Conclusion: Building upon the previous experiment, this focuses on evaluating postfix expressions using a stack ADT. It demonstrates the simplicity and efficiency of evaluating postfix expressions compared to infix expressions, showcasing the practical application of stack operations.