To understand the concepts of postfix and prefix notations, their evaluation algorithms, and implement them using C programs.

Absolutely, let's delve into the fascinating realm of postfix and prefix expressions.

Aim

- To understand the concepts of postfix and prefix notations, their evaluation algorithms, and implement them using C programs.

Theory

- **Infix, Postfix, and Prefix Notations:**
  - **Infix:** The conventional way of writing expressions where the operator is placed between the operands (e.g., `A + B`).
  - **Postfix:** Also known as Reverse Polish Notation (RPN), the operator is placed after the operands (e.g., `A B +`).
  - **Prefix:** Also known as Polish Notation, the operator is placed before the operands (e.g., `+ A B`).
- **Advantages of Postfix and Prefix:**
  - **No Ambiguity**: Eliminates the need for parentheses or operator precedence rules.
  - **Easy Evaluation**: Can be efficiently evaluated using a stack.

Procedure

1. Postfix Evaluation

- **Algorithm:**
  1. Initialize an empty stack.
  2. Scan the postfix expression from left to right.
  3. If the scanned character is an operand, push it onto the stack.
  4. If the scanned character is an operator, pop two operands from the stack, apply the operator, and push the result back onto the stack.
  5. Repeat steps 3 and 4 until the entire expression is scanned.
  6. The final result is the top element of the stack.
- **Code:**

C

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

#define MAX_SIZE 100

// Function to perform arithmetic operations
double compute(char symbol, double op1, double op2) {
    switch (symbol) {
        case '+': return op1 + op2;
        case '-': return op1 - op2;
        case '*': return op1 * op2;
        case '/':
            if (op2 == 0) {
                printf("Error: Division by zero!\n");
                exit(1);
            }
            return op1 / op2;
        case '^': return pow(op1, op2);
        default:
            printf("Error: Invalid operator '%c'!\n", symbol);
            exit(1);
    }
}

// Function to evaluate a postfix expression
double evaluate_postfix(char* expression) {
    double stack[MAX_SIZE];
    int top = -1;

    for (int i = 0; expression[i] != '\0'; i++) {
        if (isdigit(expression[i])) {
            // Push operand onto the stack
            stack[++top] = expression[i] - '0';
        } else if (isspace(expression[i])) {
            continue; // Ignore whitespace
        } else {
            // Operator encountered
            if (top < 1) {
                printf("Error: Invalid postfix expression!\n");
                exit(1);
            }
            double op2 = stack[top--];
            double op1 = stack[top--];
            double result = compute(expression[i], op1, op2);
            stack[++top] = result;
        }
    }

    if (top != 0) {
        printf("Error: Invalid postfix expression!\n");
        exit(1);
    }

    return stack[top];
}

// Function to evaluate a prefix expression
double evaluate_prefix(char* expression) {
    double stack[MAX_SIZE];
    int top = -1;
    int len = strlen(expression);

    for (int i = len - 1; i >= 0; i--) {
        if (isdigit(expression[i])) {
            // Handle multi-digit operands
            double operand = 0;
            int placeValue = 1;
            while (i >= 0 && isdigit(expression[i])) {
                operand += (expression[i] - '0') * placeValue;
                placeValue *= 10;
                i--;
            }
            i++; // Adjust for the extra decrement in the loop
            stack[++top] = operand;
        } else if (isspace(expression[i])) {
            continue; // Ignore whitespace
        } else {
            // Operator encountered
            if (top < 1) {
                printf("Error: Invalid prefix expression!\n");
                exit(1);
            }
            double op1 = stack[top--];
            double op2 = stack[top--];
            double result = compute(expression[i], op1, op2);
            stack[++top] = result;
        }
    }

    if (top != 0) {
        printf("Error: Invalid prefix expression!\n");
        exit(1);
    }

    return stack[top];
}

int main() {
    char expression[MAX_SIZE];

    printf("Enter a postfix expression: ");
    fgets(expression, MAX_SIZE, stdin);
    printf("Postfix result: %.2f\n", evaluate_postfix(expression));

    printf("Enter a prefix expression: ");
    fgets(expression, MAX_SIZE, stdin);
    printf("Prefix result: %.2f\n", evaluate_prefix(expression));

    return 0;
}
```

2. Prefix Evaluation

- **Algorithm:**
  1. Initialize an empty stack.
  2. Scan the prefix expression from right to left.
  3. If the scanned character is an operand, push it onto the stack.
  4. If the scanned character is an operator, pop two operands from the stack, apply the operator, and push the result back onto the stack.
  5. Repeat steps 3 and 4 until the entire expression is scanned.
  6. The final result is the top element of the stack.
- **Code:** (Illustrative, would require modifications to the existing code)

C

```
// ... (Similar structure to postfix evaluation, but scanning from right to
left and adjusting operand order in compute)
```

Learning Outcomes

- **Understanding different expression notations:** Gaining insights into infix, postfix, and prefix notations.
- **Stack applications**: Learning how stacks are effectively used for expression evaluation.
- **Algorithm implementation**: Translating algorithms into working C code.
- **Error handling**: Practicing how to include error checks for robust code.

Let me know if you'd like a full implementation of the prefix evaluation code or want to explore specific modifications or enhancements!