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

#define MAX_STACK_SIZE 100

typedef struct {
    int data[MAX_STACK_SIZE];
    int top;
} OperandStack;

typedef struct {
    char data[MAX_STACK_SIZE];
    int top;
} OperatorStack;

int precedence(char op) {
    switch (op) {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    }
    return 0;
}

void operandPush(OperandStack* stack, int value) {
    if (stack->top >= MAX_STACK_SIZE - 1) {
        printf("Operand Stack Overflow\n");
        exit(EXIT_FAILURE);
    }
    stack->data[++stack->top] = value;
}

int operandPop(OperandStack* stack) {
    if (stack->top == -1) {
        printf("Operand Stack Underflow\n");
        exit(EXIT_FAILURE);
    }
    return stack->data[stack->top--];
}

void operatorPush(OperatorStack* stack, char op) {
    if (stack->top >= MAX_STACK_SIZE - 1) {
        printf("Operator Stack Overflow\n");
        exit(EXIT_FAILURE);
    }
    stack->data[++stack->top] = op;
}

char operatorPop(OperatorStack* stack) {
    if (stack->top == -1) {
        printf("Operator Stack Underflow\n");
        exit(EXIT_FAILURE);
    }
    return stack->data[stack->top--];
}

void printStacks(OperandStack operand_stack, OperatorStack operator_stack) {
    printf("Operand Stack: ");
    for (int i = 0; i <= operand_stack.top; i++) {
        printf("%d ", operand_stack.data[i]);
    }
    printf("\n");

    printf("Operator Stack: ");
    for (int i = 0; i <= operator_stack.top; i++) {
        printf("%c ", operator_stack.data[i]);
    }
    printf("\n\n");
}


int eval_infix(char* expression, int start, int end) {
    OperandStack operand_stack = { .top = -1 };
    OperatorStack operator_stack = { .top = -1 };

    int i;
    int num = 0;
    int operand1, operand2;
    char operator;

    for (i = start; i <= end; i++) {
        if (expression[i] == ' ') {
            continue; // ê³µë°± ë¬´ì\213\234
        }
        else if(expression[i] >= '0' && expression[i] <= '9'){
            num = 0;
            while (i <= end && (expression[i] >= '0' && expression[i] <= '9')) {
                num = num * 10 + (expression[i] - '0');
                i++;
            }
            i--;
            operandPush(&operand_stack, num);
            printStacks(operand_stack, operator_stack); // ì\212¤í\203\235 ì¶\234ë ¥
        }
        else if (expression[i] == '(') {
            int count = 1;
            int j = i + 1;
            while (j <= end) {
                if (expression[j] == '(') {
                    count++;
                }
                else if (expression[j] == ')') {
                    count--;
                    if (count == 0) {
                        break;
                    }
                }
                j++;
            }
            operandPush(&operand_stack, eval_infix(expression, i + 1, j - 1));
            i = j;
            printStacks(operand_stack, operator_stack); // ì\212¤í\203\235 ì¶\234ë ¥
        }
        else if (expression[i] == '+' || expression[i] == '-' || expression[i] == '*' || expression[i] == '/') {
            // í\230\204ì\236¬ ì\227°ì\202°ì\236\220ì\235\230 ì\232°ì\204 ì\210\234ì\234\204ê°\200 ì\212¤í\203\235ì\227\220 ì\236\210ë\212\224 ì\227°ì\202°ì\236\220ë³´ë\213¤ ì\236\221ì\235\204 ê²½ì\232°ì\227\220ë§\214 ê³\204ì\202°
            while (operator_stack.top != -1 && precedence(expression[i]) <= precedence(operator_stack.data[operator_stack.top])) {
                operand2 = operandPop(&operand_stack);
                operand1 = operandPop(&operand_stack);
                operator = operatorPop(&operator_stack);

                int result;
                switch (operator) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    result = operand1 / operand2;
                    break;
                }
                operandPush(&operand_stack, result);
                printStacks(operand_stack, operator_stack); // ì\212¤í\203\235 ì¶\234ë ¥
            }
            operatorPush(&operator_stack, expression[i]);
            printStacks(operand_stack, operator_stack); // ì\212¤í\203\235 ì¶\234ë ¥
        }
    }

    // ë\202¨ì\235\200 ì\227°ì\202°ì\236\220 ì²\230ë¦¬
    while ((operator_stack.top != -1)) {
        operand2 = operandPop(&operand_stack);
        operand1 = operandPop(&operand_stack);
        operator = operatorPop(&operator_stack);

        int result;
        switch (operator) {
        case '+':
            result = operand1 + operand2;
            break;
        case '-':
            result = operand1 - operand2;
            break;
        case '*':
            result = operand1 * operand2;
            break;
        case '/':
            result = operand1 / operand2;
            break;
        }
        operandPush(&operand_stack, result);
        if (operator_stack.top != -1){
        printStacks(operand_stack, operator_stack); // ì\212¤í\203\235 ì¶\234ë ¥
        }
    }


    return operand_stack.data[operand_stack.top];
}

int main() {
    char infix[100];

    printf("Enter the infix expression: ");
    scanf("%[^\n]s", infix);

    int length = 0;
    while (infix[length] != '\0'){
            length ++;
    }
    length -= 1;

    int result = eval_infix(infix, 0, length);
    printf("Result: %d\n", result);

    return 0;
}
```