```c
/*
 * calculator.c
 *
 * A postfix calculator service w/ main driver. Takes a valid postfix or infix expression and outputs its sum.
 * - Limited to 8 digit numbers.
 *
 * Operations:
 * - int main()
 * - - Driver which implements the service. Runs upon execution of binary file.
 * - readInput()
 * - - reads user input into queue structure
 * - isOperator()
 * - - returns T/F determined by whether given char array is operator
 * - sumPostFix()
 * - - takes postfix queue and outputs sum
 * - convertInfixToPostfix()
 * - - when given infix queue, populates postfix queue with converted format
 * - evaluateOperatorPrecedence()
 * - - evaluates the precedence of a given operator and returns it
 * - topOperatorPrecedence()
 * - - returns true if given operator has lower precedence than top of stack, false otherwise
 * - askInputType()
 * - - asks user if input is postfix or infix format, determines driver functioning
 *
 *
 * Data Structures:
 * - memory-managed queue w/ ptr returns
 * - memory-managed stack w/ ptr returns
 *
 * by G. C. Osborn 04/24/2023
 */

/*
 * Imports
 */
# include "calculator.h"
# include "stack.h"
# include "queue.h"
# include <stdio.h>
# include <stdlib.h>
# include <string.h>

/*
 * main()
 * - Driver which implements the service. Runs upon execution of binary file.
 *
 * Parameters:
 * - NONE
 *
 * Returns:
 * - TRUE (1) - program executed successfully.
 *
 */
int main(){
    queueNode *inputQueuePtr;
    queueNode *postfixQueuePtr;
    stackNode *stackPtr;
    inputQueuePtr = makeEmptyQueue();
    postfixQueuePtr = makeEmptyQueue();
    stackPtr = makeEmptyStack();

    // ask if infix or postfix
    int infixConversion = askInputType();

    //read input expression into a queue
    readInput(&inputQueuePtr);
```

```
    //convert infix to postfix & sum
    if (infixConversion == TRUE){
        convertInfixToPostfix(&inputQueuePtr, &postfixQueuePtr, &stackPtr);
        sumPostFix(&postfixQueuePtr, &stackPtr);
    }else{
        //sum postfix expression without infix conversion
        sumPostFix(&inputQueuePtr, &stackPtr);
    }

    return TRUE;
}

/*
 * askInputType()
 * - Asks user if input is postfix or infix format, determines driver functioning.
 *
 * Parameters:
 * - NONE
 *
 * Returns:
 * - TRUE (1) - input is postfix format
 * - FALSE (0) - input is infix format
 *
 */
int askInputType(){

    char selection;
    printf("Enter 'A' for infix and 'B' for postfix, then expression on new line: ");
    scanf("%c\n", &selection);

    if (selection == 'A'){
        return TRUE;
    } else if (selection == 'B'){
        return FALSE;
    } else{
        printf("Error: reading input failed, invalid input format; exiting...\n");
        exit(1);
    }



}

/*
 * readInput()
 * - Reads user input into a queue structure.
 * - Numbers can be up to 8 digits. No error handling in that regard.
 *
 * Preconditions:
 * - inputQueuePtr is empty
 *
 * Postconditions:
 * - inputQueuePtr is populated
 *
 * Parameters:
 * - queueNode **inputQueuePtr - pointer to queue
 *
 * Returns:
 * NONE
 *
 */
void readInput(queueNode **inputQueuePtr){

    char *whitespace = " \t\f\r\v\n";
    char *token;  /* The next token in the line. */
    char *line;   /* The line of text read in that we will tokenize. */
```

```c
    /* Allocate some space for the user's string on the heap. */
    line = malloc(200 * sizeof(char));
    if (line == NULL) {
        printf("Error: malloc failed during str allocation\n");
        exit(1);
    }

    /* Read in a line entered by the user from "standard in". */
    //printf("Enter a line of text:\n");
    line = fgets(line, 200 * sizeof(char), stdin);
    if (line == NULL) {
        printf("Error: reading input failed, exiting...\n");
        exit(1);
    }
    //printf("The input line is:\n%s\n", line);

    //str not of sufficient length
    if (strlen(line) < 3){
        printf("Error: invalid input string\n");
        exit(1);
    }

    /* Divide the string into tokens. */
    token = strtok(line, whitespace);       /* get the first token */
    int anyOperators = FALSE;
    while (token != NULL) {
        int current_is_operator = isOperator(token);
        if (current_is_operator == TRUE && anyOperators == FALSE){ /* ensures we have at least one operator */
            anyOperators = TRUE;
        }

        enqueue(inputQueuePtr, token);
        token = strtok(NULL, whitespace);     /* get the next token */

    }
    if (anyOperators == FALSE && isOperator(front(*inputQueuePtr)) == FALSE){
        printf("Error: invalid input string; no operators\n");
        exit(1);
    }

    free(line);

    return;
}

/*
 * evaluateOperatorPrecedence()
 *
 * Description:
 * - Evaluates the order precedence of the given operator.
 *
 * Preconditions:
 * - Operator evaluated must be only one character long and of supported type.
 *
 * Parameters:
 * - char* token - pointer to first element of a character array (string)
 *
 * Returns:
 * - Priority (>= 0) - precedence of token
 * - ERROR (-1) - Could not evaluate precedence
 *
 */
int evaluateOperatorPrecedence(char *token){
    int token_precedence = -1;
    switch (token[0]){
        case '+': token_precedence = 0; break;
        case '-': token_precedence = 0; break;
```

```
        case '*': token_precedence = 1; break;
        case '/': token_precedence = 1; break;
        case '%': token_precedence = 1; break;
        default: break;
    }

    return token_precedence;

}

/*
 * topOperatorPrecedence()
 *
 * Description:
 * - Evaluates whether given operator has lower precedence than top of stack.
 *
 * Preconditions:
 * - Operator evaluated must be only one character long and of supported type.
 * - stackPtr must have a node on the stack.
 *
 * Parameters:
 * - stackNode** stackPtr - pointer to stack
 * - char* token - pointer to first element of a character array (string)
 *
 * Returns:
 * - TRUE (1) - top operator higher or equal priority than token
 * - FALSE (0) - top operator not higher priority than token
 *
 */
int topOperatorPrecedence(stackNode *stackPtr, char *token){

    int topPriority = evaluateOperatorPrecedence(top(stackPtr));
    int tokenPriority = evaluateOperatorPrecedence(token);

    if (topPriority >= tokenPriority){
        return TRUE;
    } else { // token less than
        return FALSE;
    }

}

/*
 * isOperator()
 * - returns T/F determined by whether given char array is operator.
 *
 * Parameters:
 * - char* token - first element of char array.
 *
 * Returns:
 * - TRUE (1) - token is an operator.
 * - FALSE (0) - token is not an operator.
 *
 *
 */
int isOperator(char* token){
    int flag = FALSE;
    switch (token[0]){
        case '+': flag = TRUE; break;
        case '-': flag = TRUE; break;
        case '*': flag = TRUE; break;
        case '/': flag = TRUE; break;
        case '%': flag = TRUE; break;
        default: flag = FALSE; break;
    }
    return flag;
}
```

```c
/*
 * convertInfixToPostfix()
 * - When given infix queue, populates postfix queue with converted format.
 *
 * Preconditions:
 * - Empty postFixQueue and empty stackPtr, populated and well-formed inputQueuePtr
 *
 * Postconditions:
 * - Populates the queue of postfixQueuePtr, stackPtr stack and inputQueuePtr queue empty.
 *
 * Parameters:
 * - queueNode **inputQueuePtr - queue with infix input elements
 * - queueNode **postfixQueuePtr - empty queue
 * - stackNode **stackPtr - empty stack
 *
 * Returns:
 * - NONE
 *
 */
void convertInfixToPostfix(queueNode **inputQueuePtr, queueNode **postfixQueuePtr, stackNode **stackPtr){
    printf("Converting to postfix...\n");
    char token[8];

    while (queueEmpty(*inputQueuePtr) == FALSE){

        dequeue(inputQueuePtr, token); // character to be read

        if (isOperator(token) == FALSE){ // if parenthesis or operand

            if (token[0] == '('){ // beginning of parenthesis pseudo-evaluation where operators are coupled
                push(stackPtr, token);

            } else if (token[0] == ')'){ // decoupling of operators within parenthesis set
                while (top(*stackPtr)[0] != '('){
                    char op[8];
                    pop(stackPtr, op);
                    printf("%s ", op);
                    enqueue(postfixQueuePtr, op);
                }
                char op[8];
                pop(stackPtr, op);

            } else { // add operand to postfix queue normally
                printf("%s ", token);
                enqueue(postfixQueuePtr, token);
            }

        } else {

            // pop all lower precedence elements off stack and add to postfix
            while (stackEmpty(*stackPtr) == FALSE && topOperatorPrecedence(*stackPtr, token) == TRUE){
            //not greater and not equal, eg. less than stack
                char op[8];
                pop(stackPtr, op);
                printf("%s ", op);
                enqueue(postfixQueuePtr, op);

            }
            push(stackPtr, token);
        }
    }

    // remaining operands with the exception of any initial parenthesis
    while (stackEmpty(*stackPtr) == FALSE && top(*stackPtr)[0] != '('){
        char op[8];
        pop(stackPtr, op);
```

```
        printf("%s ", op);
        enqueue(postfixQueuePtr, op);
    }
    printf(" END \n");
}


/*
 * sumPostFix()
 * - takes a postfix queue and outputs its sum to stdout.
 *
 * Preconditions:
 * - stackPtr points to an empty stack
 *
 * Postconditions:
 * - Empties inputQueuePtr queue and stackPtr stack.
 *
 * Parameters:
 * - queueNode **inputQueuePtr - pointer to a queue
 * - stackNode **stackPtr - pointer to a stack
 *
 * Returns:
 * - NONE
 *
 */
void sumPostFix(queueNode **inputQueuePtr, stackNode **stackPtr){
    printf ("BEGIN POSTFIX SUM\n");
    /* note here that we use atoi because we work with char values */

    char token[8]; // this is the read value
    while (queueEmpty(*inputQueuePtr) == FALSE){
        dequeue(inputQueuePtr, token); // read
        if (isOperator(token) == FALSE){
            push(stackPtr, token);
        }else{ // sum the operands with the operator
            int sum = 0;
            char pushSum[8];
            if (stackEmpty(*stackPtr) == TRUE){ break; }

            char op1str[8];
            pop(stackPtr, op1str);
            char op2str[8];
            pop(stackPtr, op2str);
            int op1 = atoi(op1str);
            int op2 = atoi(op2str);
            switch (token[0]){
                case '+': sum = op2 + op1; break;
                case '-': sum = op2 - op1; break;
                case '*': sum = op2 * op1; break;
                case '/': sum = op2 / op1; break;
                case '%': sum = op2 % op1; break;
            }
            sprintf(pushSum, "%d", sum);
            push(stackPtr, pushSum); // push sum back into stack
            printf("summing %s %s %s resulting in %s\n", op1str, token, op2str, pushSum);

        }
    }
    char sum[8];
    pop(stackPtr, sum); // read sum
    printf("Sum: %d\n", atoi(sum));
    return;
}
```