Worksheet: Stack ADT (Static Implementation)

This worksheet is concerned with static implementations of stacks and you will be introduced to the concept of abstract data types. You are not expected to complete the entire worksheet in class. Work on as many problems as you can; the remaining problems you can use for practice and to test and refine your understanding.

1. Implement the abstract data type Stack using an array. You may use the following declaration for a static stack:

```
struct  StackRecord
{
      int capacity;
      int topOfStack;
      int* array;
};
typedef struct StackRecord* Stack;
#define EmptyTos -1
#define MinStackSize 5
```

   The operations to be implemented include CreateStack(), MakeEmptyStack(), EmptyStack(), PushStack(), PopStack(), TopOfStack(), IsFullStack(), IsEmptyStack (), DisplayStack().

2. Write a corresponding main program to test your stack implementation by asking the user to continuously push and pop integer values to and from the stack, respectively.

3. Write a function that reads a list of integers and when user enters EOF key prints them in reverse order. This is a very simple function that creates a stack, reads a series of numbers and pushes them onto the stack. When the user keys end of file, the program pops the stack and prints the numbers in reverse order.

4. Write a C program which checks whether parentheses match, i.e. whether the number of open parentheses '(' is the same as the number of close parentheses ')'. Other characters may occur between the parentheses, but we do not process those any further. The algorithm proceeds as follows:

```
Algorithm ParseParenthesis
Loop (more data)
  read (character)
  if(opening parenthesis)
     pushStack (stack, character)
  else
    if(closing parenthesis)
      if(emptyStack(stack))
        print error - closing parenthesis not matched
      else
        popStack(stack)
      endIf
    endIf
  endIf
  if(not emptyStack(stack))
    print error - opening parenthesis not match
end ParseParanthesis
```

5. Write a function to implement the postfix evaluation. The function should read a postfix string consisting of only multi-digit numeric data and the +, -, *, and/ operators, call the evaluation algorithm, and then print the result.

6. Write a function that implements the infix to postfix notation. The function should read an infix string consisting of single alphabetic characters for variables, parentheses, and the +, -, *, and / operators; call the conversion algorithm; and then print the resulting postfix expression. Please note that the priorities of the operators are as follows:
    - Priority 2: *, /
    - Priority 1: +, -
    - Priority 0: (

```
ALGORITHM
Loop (<length of the formula)
        read one token from the formula
        if(token is parenthesis)
                pushStack(stack,token)
        else if(token is close parenthesis)
                popStack(stack, token)
                Loop(token is not open parenthesis)
                        concatenate token to postfix
                        popStack(stack, token)
                end loop
        else if(token is operator)
                stackTop(stack, topToken)
                Loop(not Empty Stack && priority(token) <= priority(topToken))
                        popStack(stack, tokenOut)
                        concatenate tonekOut to postfix
                        stackTop(stack,topToken)
                end loop
                pushStack(stack,token)
        else
                Concatenate token to postfix
        go to the next token of the formula
end loop
If(formula empty)
        pop stack and concatenate tokens to postfix
else
        Loop(notEmpty(stack))
                popStack(stack, token)
                concatenate tonekOut to postfix
        end loop
return postfix
```