Exercise 4a. (RPN-calculator; how to use a given ADT stack, 1p)

Write a RPN-calculator program. The program can be used for simple arithmetic operations for integers (they can be one digit integers if you like). The possible arithmetic operations are: + and - (addition and subtraction). The two other operators user can enter are = and Q). You have to use the stack component template programmed ready for you. The interface and the implementation are in the file stack.h, which is available from the Oma-portal.

The application you must create, reads integers and operators from the keyboard in postfix form. All numbers read are pushed onto the stack. When arithmetic operator + or - is encountered, the corresponding calculation is performed for operands popped from the stack. The result is pushed back onto the stack. The operation = in the input stream means "print the result to the screen". In practice it is: pop the topmost value from the stack and print the value to the screen (and push it back to the stack). You must be able to continue the calculation (from the previous intermediate result). That's why you have to push the printed result back to the stack. The operator Q means that the content of the stack is printed to the screen and the program terminates.

Example: If you enter the following data to your program you will get the results below:

```
This is a RPN calculator. Enter operands and operators:

     Input    Output
1
2
3
+
=      top value is 5
2
-
=      top value is 3
1
+
=      top value is 4
Q      stack contained
4
1
```

Hint. If you use >> operator to read characters, you get char type results[1]. Because you can use one digit numbers, it is easy to make conversion from character input to the integers

[1] `_getche()` function (defined in <conio.h>) reads keyboard characters one-by-one without buffering, but it is available only in Windows environment.

(int), because codes of digits 0, 1, 2, ..., 9 are correspondingly 48, 49, 50, ..., 57 in the used encoding system (UTF-8 or ASCII). This means that if you get input in the form

```
cin >> input; // input is defined as: char input;
```

then you get the corresponding int value in the following way

```
if (isdigit(input))
     number = input - 48; // or number = input - '0'
```

where the variable number is declared as: int number;. isdigit() a standard macro, and it is defined in <ctype.h>

Extra exercise 4b. (Converting infix notation to postfix notation, 0.25 bonus)

Warning! This is quite demanding exercise.

Phase 1

Make an application which converts infix-equations to postfix-format. Operands are one-digit positive integer numbers and operators are '+', '-', '*' and '/'. There are no parenthesis allowed in this phase. First the application reads one line of characters (max 80 characters) containing an infix-equation and then converts that to the matching postfix format. It is useful to implement a function `void infixTopostfix(char *infix, char *postfix)` that creates postfix equation in string from the given infix string.

One example how the application should work

```
Infix to postfix converter

Give an infix equation at one line
1+2*3
Equivivalent postfix equation is 123*+
```

Note 1: use the given stack template stack.h.

Hint 1: In the lecture we studied how the infix to postfix conversion can be done. Small recap here: Infix equation is processed from the left to the right, symbol-by-symbol. When we encounter an operand it is appended to the postfix equation directly. For handling operators we need an operator stack. When we encounter an operator (in the infix notation) we should check if there exist higher-priority operators in the stack. If there are, then operators are pop'd from the stack and appended to the postfix equation until there are no higher priority operators left on the stack. Then the just read operator is push'd to the stack. After we have scanned the whole input infix equation, remaining operators on the stack are appended to the postfix equation.

Hint 2: It would be a good idea to implement a special function for operator's precedence comparision.

Phase 2

Implement a better infix to postfix converter which allows also an infinite amount of parenthesis that control the order of evaluation. For example

```
Infix to postfix converter

Give an infix equation at one line
(((3+6+1)*(2-4))+7)
Equivivalent postfix equation is 36+1+24-*7+
```

Phase 3

Implement a calculator for one-digit integers. Modify your RPN-calculator (exercise 6a) in such a way that it can accept string as an input, and add '* and '/' operations to the RPN-calculator. This RPN-calculator function can be declared as follows `int evaluatepostfix(char *postfix)`. Then give the converted postfix string from the previous phase 2 to the RPN-calculator function and print the result, like

```
Infix to postfix converter and calculator

Give an infix equation at one line
(1+2)*(3+4)
Equivivalent postfix equation is 12+34+*
And it's value is 21
```