

Exercise Postfix Calculator (`postfix.c`)

Implement a postfix calculator for integer numbers. A postfix calculator reads expressions in postfix notation. In this notation, the function symbol in an expression is notated behind the arguments. For instance, the infix expression $1 + 4$ corresponds to $(1, 4) +$ in postfix notation. We will omit the syntactic sugar and write $1\ 4\ +$.

Expressions in postfix notation can be evaluated using a stack. For this exercise, implement a stack using a linked list and use it to store numbers. The stack should be stored in a separate header and implementation file (`stack.c`, `stack.h`).

You should read a mixture of integer numbers and operations (separated by white characters). Once = your program should print the result of the calculation and terminate. You may assume that an entered word will consist of at most 10 characters. To convert a string into a number you can use `strtol` or `sscanf`. You may assume that there are only correct inputs.

Your program should support the following operations: $+$, $-$, $*$, $/$ and $\%$.

Example:

```
$ ./postfix
1 2 + =
3

$ ./postfix
11 2 3 * + =
17
```

Read the next sections for more details.

Prefix notation, infix notation and postfix notation

In mathematics and also programming, there are several ways of denoting an expression. In most programming languages like C we usually use the prefix notation. In this notation, the function that is applied is written before the arguments. For example, an application of the function `atoi` is written like `atoi(s)`, `printf` and `scanf` are written in a similar way. In mathematics, the prefix notation is also often used, e.g., the function sine is usually applied in this way: $\sin \frac{\pi}{4}$. Also the negation $-x$ is a prefix notation. This notation is also called polish notation (after the mathematician Jan Lukasiewicz).

For binary functions (functions with two arguments like addition or multiplication) we are very used to the infix notation in which the function is written in between the arguments: $1 + 2$ or $7 * 3$. In prefix notation, this would correspond to `add(1, 2)` or `mul(7, 3)` or also $+ 1\ 2$. In C, the ternary operator `cond ? then : else` is also in infix notation.

Finally, there is the not that common postfix notation, sometimes called reverse polish notation. In this notation, the function is written after the arguments. In mathematics, a few functions are written in this way, like factorial ($5! = 5 * 4 * 3 * 2 * 1$). The expression $1 + 2$ is written as $12+$ in postfix notation.

Prefix and postfix notation have the advantage that no brackets or precedence rules are needed (if all function symbols have a fixed number of arguments, which we assume here - see http://en.wikipedia.org/wiki/Reverse_Polish_notation for more details). Furthermore, postfix notation can be easily evaluated using a stack.

Stack

A stack is a data structure with two operations:

- `push` adds an element to the top of a stack

- **pop** removes and returns the top element of a stack

A stack hence can be easily implemented using a single linked list. The push operation corresponds to adding an element to the top of the list while **pop** removes the first element and returns its value.

Postfix calculator

We can evaluate postfix expressions in a simple program:

- Read the next token (number or operation)
 - If it is a number, push the number onto the stack
 - If it is an operation, pop the arguments from the stack (be careful that the order is important for subtraction and division), apply the operation and push the result back onto the stack.
 - At the end, there is only one element left on the stack - the solution.

Consider the calculation $(5 - 3) * (2 + 4)$. Converting $5 - 3$ to postfix notation we obtain $5\ 3\ -$, $2 + 4$ is $2\ 4\ +$ and hence the whole expression in postfix notation is $5\ 3\ -\ 2\ 4\ +\ *$.

Observe that if we skip the brackets in infix notation we obtain $5 - 3 * 2 + 4$ which is usually read as $(5 - (3 * 2)) + 4$ and corresponds to $5\ 3\ 2\ * \ -\ 4\ +$ in postfix notation.

Evaluating $5\ 3\ -\ 2\ 4\ +\ *$ yields the following steps:

1. 5: push onto stack: [5]
2. 3: push onto stack: [5, 3]
3. -: pop second argument (3) and then first argument (5) from the stack, subtract them and push the result 2 back onto stack: [2]
4. 2: push onto stack: [2, 2]
5. 4: push onto stack: [2, 2, 4]
6. +: pop arguments from stack, add them and push the result onto the stack: [2, 6]
7. *: pop arguments from stack, add them and push the result onto the stack: [12]

Hence, we finally obtain the correct result 12.

Evaluating the second expression $5\ 3\ 2\ * \ -\ 4\ +$ yields the following steps:

1. 5: push onto stack: [5]
2. 3: push onto stack: [5, 3]
3. 2: push onto stack: [5, 3, 2]
4. *: pop arguments from stack, multiply them and push the result onto the stack: [5, 6]
5. -: pop arguments, subtract them and push the result onto the stack: [-1]
6. 4: push onto stack: [-1, 4]
7. +: pop arguments from stack, add them and push the result onto the stack: [3].

We now obtain the result 3.

Detailed exercise

Create at least three files: **stack.h**, **stack.c** and **postfix.c**. The latter of these files should contain the **main**-procedure. The **stack**-files should contain the prototypes and implementations of the stack.