Practical Documentation CS2002 W7

Reverse Polish Calculator in C.

Student ID: 180023970

Submission Date: 10/03/21
Tutor: Susmit Sarkar

Overview

In this practical, I was instructed to develop a Stack ADT based on array and a Calculator that accepts a Reverse Polish Notation in C. It was designed to provide experience with testing, working with structs, pointer and most importantly building a C program from modular functions.

I achieved all required functionality, providing a thoroughly tested Stack ADT implementation, as well as a functioning RPN Calculator.

This report is going to start by discussing code design decisions and testing, and finish with a conclusion.

Design

The program is implemented using statically allocated memory, to reduce complications in code.

Stack

The stack implementation did not involve any substantial design decisions, as the function prototypes were provided. The array powered stack functionality worked by moving the “top” pointer on pop / push along the array, thus fulfilling the last in first out principle (pushed is at top -> popped removes at top).

Calculator: calculator logic

Developing the logic of the calculator, I tried to maintain that the function serves a single purpose. Balancing this with not over decomposing code should have helped with code readability & debuggability. To aid this, I tried to visualize the processes, and develop my code accordingly to it. Please see the Figure 1 below.

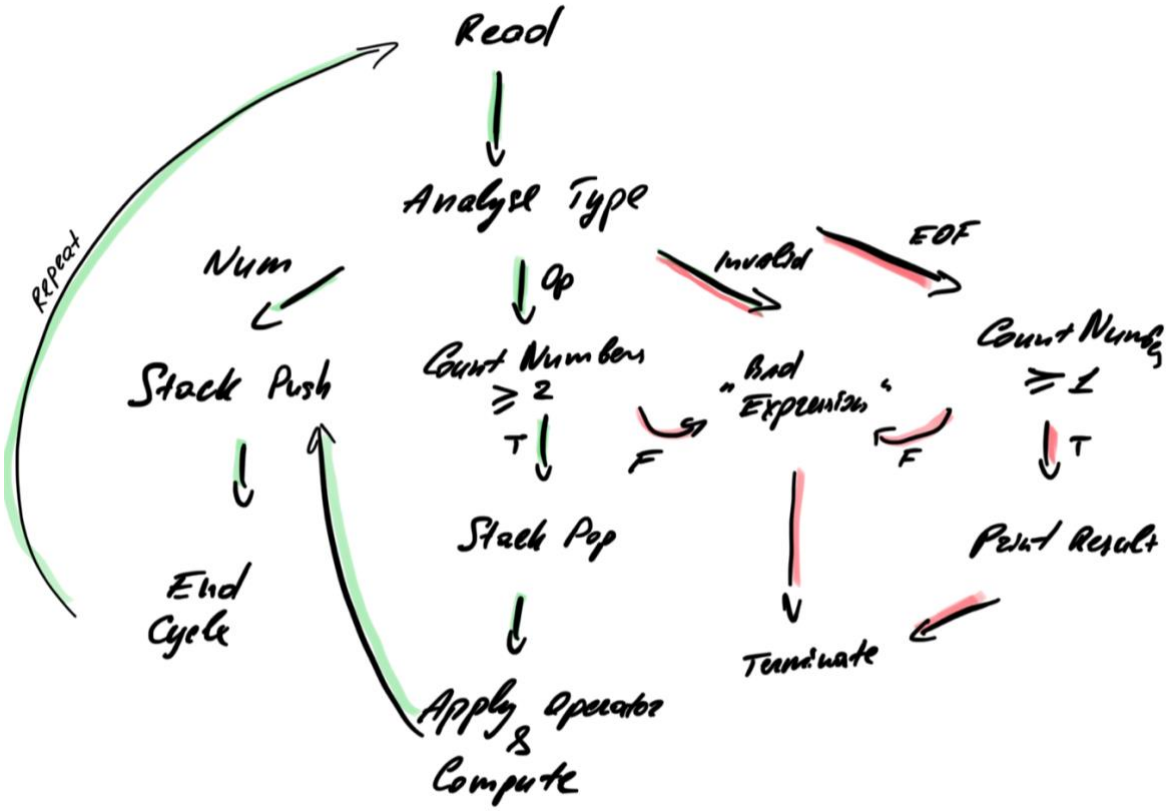

Figure 1

In my implementation, I started with classifying input in types, to aid control flow. This is achieved using an enumeration and a function that processes an input char and returns its type : `inputTypeOf(int input)`

I decided to group logic of processing operator input & terminating program together in container functions, as in the practical spec I was advised to keep the main method as a series of function calls.

Handling operator processing involved checking for negative numbers, processing compute functions and determining if the expression is invalid prior to computing. I attempted to group logic inside the function to resemble the single purpose of a function principle.

Terminating the program was grouped in a way to reduce repetition of code and seemed intuitive from the graphic representation of the processes.

Calculator: main method

The main methods consists mainly of the control flow logic around reading the input and deciding on what should be called to process it. It is done so by using the switch statement iterating through input types of the input & applying respective functions.

Calculator: helper functions

To improve code readability and increase abstraction from lower lever operations, helper functions were written. In particular, the processes around reading input were broken down into higher level function, such as `peekInput`

Testing

I tested the stack implementation with 5 custom methods that tested core stack operations & corner cases. I relied mainly on the stacscheck for ensuring that the calculator works, as well as manual testing.

```
Testing CS2002 W07-C2
- Looking for submission in a directory called 'src': found in current directory
* BUILD TEST - build-clean : pass
* BUILD TEST - public/Test01_RunYourStackTests/build : pass
* INFO - public/Test01_RunYourStackTests/infoTest : pass
--- submission output ---
Stack Tests complete: 7 / 7 tests successful.
---------------
---

* BUILD TEST - public/Test02_RunCalculatorTests/build : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-0.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-1.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-2.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-3.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-bad-1.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-bad-2.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-bad-3.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-bad-4.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-float-1.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-float-2.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-longer-1.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-negative-1.out : pass
* COMPARISON TEST - public/Test02_RunCalculatorTests/prog-negative-2.out : pass
17 out of 17 tests passed
```

Figure 2

As can be seen on Figure 2, my 5 methods + 2 provided for stack pass, as well as the stacscheck provided functions for builds & calculator functionality. Please see screenshots for manual testing below.

```
2 3 - 3 - 4
Bad expression
```

```
2 - 3
Bad expression
```

```
- 2 3
Bad expression
```

```
1 -3 * 9 3 /
Bad expression
```

```
1 -3 * 9 3 / +
0.00
```

```
1 -3 + -4 2 + *
4.00
```

```
45 44 - 25 4 * *
100.00
```

Conclusion

I believe my solution comes up to the functional requirements of the practical, holding the stated simplifications true (fixed stack size with no expansion; spaces between inputs; 4 operators).

Given more time, I would have automated calculator testing and designed inputs properly to tackle corner cases. In addition, I believe my decomposition could have been improved, particularly in the main method of the calculator, to aid in readability.

Overall, this practical has been a good challenge to get experience with C I/O handling, working with pointers, structs and building programs in C.