LW: Stack Calculator

Overview

Objectives

- Correctly code a stack implementation using step-by-step pseudocode.
- Correctly code a stack-based mathematical algorithm using step-by-step pseudocode.
- Correctly apply memory management steps for an algorithm that uses the heap (resizing an array).
- Use I/O streams to properly read user input and output a correct arithmetic result.

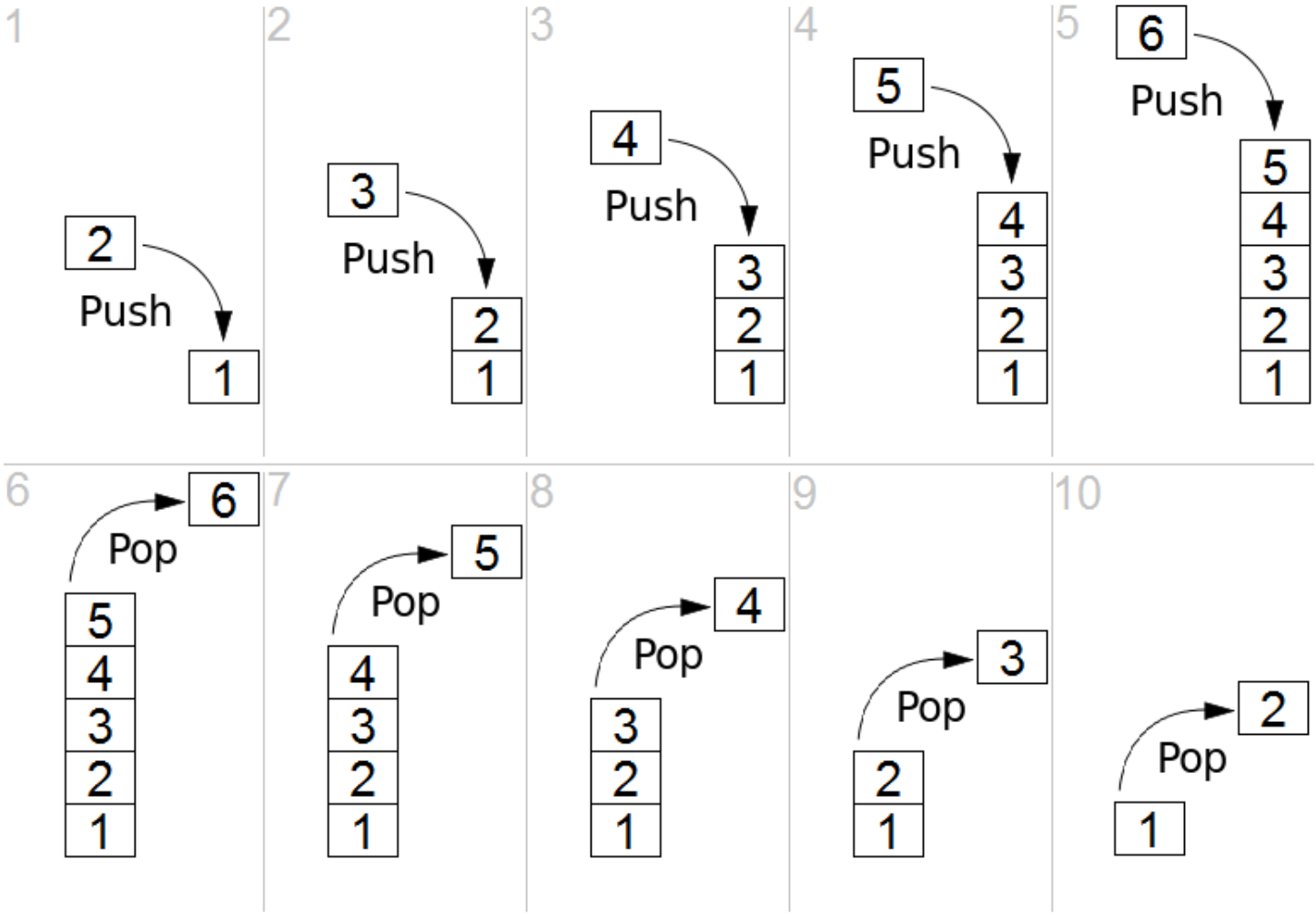
Completion

- Attend lab session or have an excused absence
- Get 45 points on Gradescope

Getting Started

- Get the starter code (the files you will submit).
    - `"functions.cpp"`
        - You should write task 1 here.
    - `"functions.h"`
        - You should read this.
    - `"calculator.cpp"`
        - You should write task 2 here.
- Only task 1 is required.
- Allowed includes:
    - `<iostream>`
    - `<string>`
    - `<sstream>`
    - `"functions.h"`
- Your program should not leak memory
    - Every `new` should have a corresponding `delete` (or `delete[]`).
    - Compile with `-fsanitize=address,undefined`.

Background Information



Simple representation of a stack runtime with push and pop operations (view Wikipedia source.). In this example, the stack is currently represented as an integer array with a single element of value 1.

- In Step 1, the value 2 is added to the stack in an action called a push.
- In Steps 2 through 5, the values 3 through 6 are pushed to the stack one-by-one.
- In Step 6, the value 6 is removed from the array in an action called a pop.
- In Steps 7 through 10, the values 5 through 2 are popped from the stack one-by-one.

A ***stack*** is an abstract data type that serves as a collection of elements and has two principal operations:

1. **push**: add an element at the top of the stack
2. **pop**: remove the element at the top of the stack, returning the removed value. (If nothing is in the stack return INT32_MAX)
3. **peek**: give access to the top element

Stack's follow a "last in, first out" principle (LIFO).

Requirements

The labwork consists of two tasks.

Task 1 (required): Implement a Stack (of `ints`)

Read `functions.h`, which contains the definition of the Stack type and declarations of the Stack operations which you must implement (`push()`, `pop()`, and `peek()`).

Pushing and popping should be done at the back (last element) of the array:

- `Let numbers = [1, 2, 3,  ], capacity = 4, count = 3`
- `Push 4`
- `Now numbers = [1, 2, 3, 4], capacity = 4, count = 4`
- `Pop`
- `(returns 4)`
- `Now numbers = [1, 2, 3, _], capacity = 4, count = 3`
- `Push 5`
- `Now numbers = [1, 2, 3, 5], capacity = 4, count = 4`

Resizing the array should **double** the capacity:

- `Let numbers = [1, 2, 3, 4], capacity = 4, count = 4`
- `Push 5`
- `Now numbers = [1, 2, 3, 4, 5, _, _, _], capacity = 8, count = 5`

Task 2 (optional): Implement a 4-function RPN Calculator

Reverse Polish Notation (RPN) is a mathematical notation in which operators follow their operands (also called *postfix notation*).

- For example, the arithmetic expression `"1 + 2"` would be written as `" 1 2 +"` in RPN.

One of the benefits of RPN is that parentheses are not required.

- For example, the arithmetic expression `"1 + 2 * 3 - 4"` is ambiguous without some assumption about the order of operations.

According to the standard order of operations (PEMDAS), the expression means:

PEMDAS	RPN
`(1 + ((2 * 3) * 4)) - 5`	`1 2 3 * 4 * + 5 -`
`(1 + (6 * 4)) - 5`	`1 6 4 * + 5 -`

`(1 + 24) - 5`	`1 24 + 5 -`
`25 - 5`	`25 5 -`
`20`	`20`

The RPN algorithm is simple and uses a stack to store intermediate values of the computation:

1. Read numbers and push onto the stack until you read an operator: +, −, *, =, ^
2. If an operator is read,
    1. pop the top two elements of the stack into variables for right and left operands
        1. First stack pop operation is stored as the right operand
        2. Second stack pop operation is stored as the left operand
    2. Do the operation and push the result onto the stack
3. Repeat steps 1 and 2 until `'='` is read
4. Pop the result from the stack

You should read from standard input and print to standard output.

Examples for RPN Calculator

Input	**Expected Output**
`1 2 + 3 * 4 - =`	`5`
`-5 -7 -9 * - =`	`-68`
`10 10 10 * * 59 + 1024 8 * * 9 - 1000000 - =`	`7675319`
`1 2 + 3 4 + + 5 6 + 7 8 + + + 9 10 + + =`	`55`
`1 2 3 4 5 6 7 8 9 10 + + + + + + + + + =`	`54`
`100 50 - 25 - 12.5 - 6.25 - 12.5 - =`	`-5`
`1 2 3 4 5 6 7 8 9 * * * * * * * * * =`	`362880`
`4 2 3 ^ ^ =`	`65536`

use

1, 2, +

1 goes to stack

2 goes to stack

Pop -> 2 -. Right

Pop -> 1 -> left

10 11 15