Data Structures and Algorithms — Lab 3

# Objective

* Implementation and application of Stacks

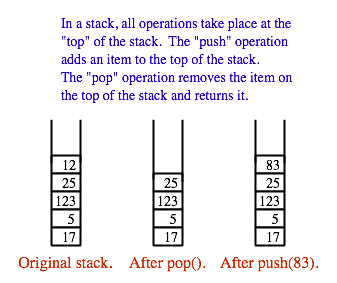
# Introduction

Think about a stack of books. If you have a stack of books sitting on your desk, you can "push" another book on top, or you can "pop" one off the top. Trying to grab one from the middle or bottom is difficult. You're much safer trying to "push" and "pop" from the top of this stack of books.

All of the operations of a stack operate on the one end of the stack, known as the **top** of the stack. If you want to access the second item from the top, you're out of luck (unless you **pop()** the first item). If you want to access the item at the bottom of the stack, you're out of luck (unless you **pop()** *everything* else off first). If you're using a stack, you get access only to the top.

That we need to provide six operations to implement a stack. These operations are:

1. **Stack(int)** //initialize stack values and parameters using a **CONSTRUCTOR**
2. **isEmpty( )** // no element in stack
3. **isFull( )** // size of stack = number of elements in stack
4. **push( )** //add an element from stack (always on top)
5. **pop( )** //remove an element from top of stack after
6. **~Stack( ) //** Delete dynamically allocated memory using a **DESTRUCTOR**



# Lab Task 1

Apply and test the simple STACK implementation using Arrays and templates below:

* Stack(int size) // Allocate a dynamic memory of given size and initialize other parameters
* void Push(GEN value) // value saved on top of stack
* GEN Pop () // return top value and decrease the top by one
* bool isEmpty() // check the stack is empty or not
* bool isFull() // check the stack is full or not
* void Display() // display the whole stack from top to bottom
* GEN Top() // return the top value but do not pop/remove it from the stack
* ~Stack() // Delete dynamically allocated memory

**Your program should be menu based only – create the menu give below:**

1. **Initialize the stack**
2. **Push into stack**
3. **Pop from stack**
4. **Display Top value**
5. **Display contents of complete stack**
6. **Destroy the stack**
7. **Exit**

# Lab Task 2

A bracket is considered to be any one of the following characters: (,), {,}, [, or].

Two brackets are considered to be a matched pair if the opening bracket (i.e., (, [, or {) occurs to the left of a closing bracket (i.e.,),], or}) of the exact same type. There are three types of matched pairs of brackets: [], {}, and ().

A matching pair of brackets is not balanced if the set of brackets it encloses are not matched. For example, {[(])} is not balanced because the contents in between {and} are not balanced. The pair of square brackets encloses a single, unbalanced opening bracket, (, and the pair of parentheses encloses a single, unbalanced closing square bracket].

## Implementation

You have to implement the following functionality in stack which takes input of character array and checks either the array has matching pair brackets or not.

## Sample Input:

1. [[{(())}]]
2. {{[[])}

## Sample Output:

1. Expression is balanced
2. Expression is not balanced

# Lab Task 3

Reverse contents of a stack in main program. Contents of the original stack must be reversed. You are not allowed to use any data structure other than stack (arrays in main cannot be used) – more than one stack can be used if required. You are not allowed to modify any function of the stack or add a new function in the stack. This reversal has to be done in the main program.

# Lab Task 4

You are supposed to write a code that takes a **Decimal Number** as input and shows its **Binary Form** on the screen using stacks.

**Note:** You have to make the logic **on your own**. Consider it as your evaluation over Stacks.

## Sample Input:

**9**

## Sample Output:

**1001**