

CC-213L

Data Structures and Algorithms

Laboratory 04

Stack ADT

Version: 1.0.1

Release Date: 09-10-2024

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

- Learning Objectives
- Required Resources
- General Instructions
- Background and Overview
 - Pointers and Dynamic memory allocation
 - Data Structures
 - Types of Data Structures
 - Stack ADT
 - Types of Stack ADT
- Activities
 - Pre-Lab Activity
 - Task 01
 - Stack ADT I
 - Stack ADT II
 - Task 02: Balanced Parenthesis
- References and Additional Material

Learning Objectives:

- Pointers and Dynamic memory allocation
- Stack Data Structure
- Using the concepts of Dynamic Memory Allocation
- Standard Operations on the Stack Data Structure
- Use of Stack ADT in Different Applications

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how s/he is doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the following.

Teachers:		
Course / Lab Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistant	Muhammad Tahir Mustafvi	bcsf20m018@pucit.edu.pk
	Maryam Rasool	bcsf21m055@pucit.edu.pk

Background and Overview:

Pointers and Dynamic memory allocation

In C++, pointers and dynamic memory allocation are fundamental concepts for managing memory and creating more flexible data structures. Here's an overview of pointers and dynamic memory allocation in C++:

Pointers:

Pointer Basics: A pointer is a variable that stores the memory address of another variable. It "points" to the location in memory where the data is stored. In C++, you declare a pointer using an asterisk (*) before the variable name.

Dynamic Memory Allocation:

C++ provides the `new` operator for allocating memory dynamically on the heap and the `delete` operator for freeing that memory when it's no longer needed.

Example:

```
--
37  int main()
38  {
39      int* ptrToInt = nullptr;
40      double* ptrToDouble = nullptr;
41      char* ptrToChar = nullptr;
42      ptrToInt = new int;
43      ptrToDouble = new double;
44      ptrToChar = new char;
45      *ptrToInt = 10;
46      *ptrToDouble = 20.02;
47      *ptrToChar = 'A';
48      cout << *ptrToInt << endl;
49      cout << *ptrToDouble << endl;
50      cout << *ptrToChar << endl;
51      delete ptrToInt; delete ptrToDouble; delete ptrToChar;
52
53      return 0;
54  }
```

Figure 1 (pointer and dynamic memory)

Explanation:

This C++ program demonstrates the allocation of memory for an integer, a double, and a character. In line 39, we declare a pointer to an integer, and in lines 40 and 41, we declare pointers to a double and a character, respectively. Starting from line 42 and extending to line 44, we utilize the `new` operator to reserve memory for these variables dynamically. Subsequently, in lines 45 to 47, we initialize these memory locations with specific values. After displaying the values, we proceed to deallocate all of these memory allocations. The `delete` operator is employed for this purpose, ensuring that the allocated memory is released properly.

Output:



```
Select Microsoft Visual Studio Debug Console
10
20.02
A
```

Figure 2 (Pointers and dynamic memory allocation)

Memory Management Best Practices:

- Always free dynamically allocated memory using `delete` or `delete[]` to avoid memory leaks.

- Be cautious when using raw pointers, as they can lead to bugs like null pointer dereferences and memory leaks. Consider using smart pointers when possible.
- Avoid allocating excessive memory on the heap, as it can lead to fragmentation and reduced program performance.
- Practice good memory management by deallocating memory as soon as it's no longer needed.

Data Structure:

Data structure refers to the way data is organized, stored, and accessed in a computer system. It provides a systematic way of managing and manipulating data efficiently. Data structures are designed to optimize operations such as insertion, deletion, searching, and sorting of data.

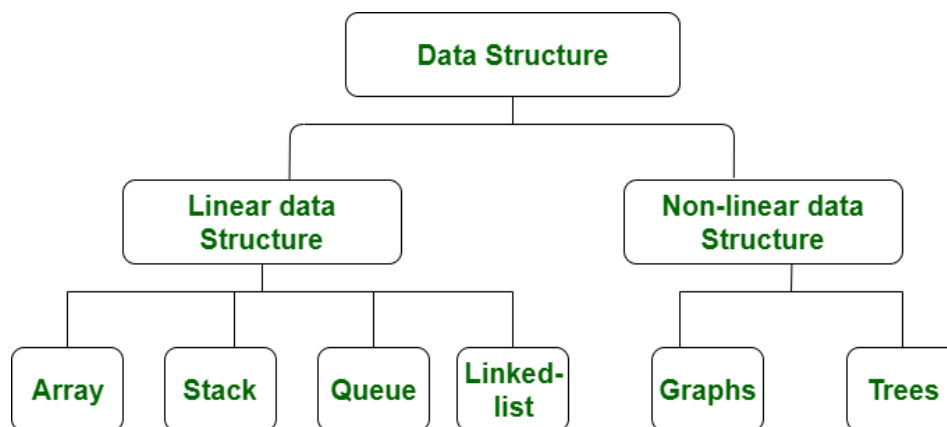
Types of Data Structure:

Figure 3 (Types of Data structures)

Linear Data Structures:

Linear data structures store data elements sequentially, one after another, in a linear fashion. They follow a specific order or sequence. Examples include arrays, linked lists, stacks, and queues.

Non-linear Data Structures:

Non-linear data structures do not store elements in a sequential manner. They allow elements to be connected in multiple ways, forming a hierarchical or interconnected relationship. Examples include trees and graphs.

Stack ADT:

A stack is a linear data structure that follows the **Last-In-First-Out** (LIFO) principle. It can be visualized as a stack of plates, where the last plate placed on top is the first one to be removed. The two main operations on a stack are **push** (to add an element to the top) and **pop** (to remove the top element). The element below the top is inaccessible until the top element is removed. A basic example of a stack is the call stack in programming, which keeps track of function calls and their local variables. Another example is the Undo feature in text editors, where the most recently performed action can be undone by popping it from the stack.

Types of Stacks:**Fixed Size Stack:**

As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.

Demonstration:

```

37     template <class T>
38     class Stack
39     {
40     :   T* stk[10];
41     :
42     :   // rest implementation of Stack
43     :
44     };

```

Figure 4 (Fixed sized Stack)

Explanation:

Line 40 statically allocates a fixed-size stack using an array of 10 elements. Each time an object of the "Stack" class is created, it will have a pre-allocated array of 10 elements in memory, and this size remains constant.

Dynamic Size Stack:

A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, or dynamically allocated Array as it allows for easy resizing of the stack.

Demonstration:

```

37     template <class T>
38     class Stack
39     {
40     :   T* stk;
41     public:
42     :   Stack(int size=10):stk(new T[size])
43     :   {
44     :       // rest implementation
45     :   }
46     :
47     :   // rest implementation of Stack
48     :
49     };

```

Figure 5 (Dynamic Stack)

Explanation:

In line number 40 there is pointer to generic type. In constructor we have allocated memory of 10 size array. Stack of any size can be allocated at run time by passing any size of array to the constructor of Stack. In below figure in line 66 Stack of size 10 is allocated which was the default size. In line 67 Stack of 20 size is allocated and at line number 68 Stack of size 100 is allocated.

```

66     Stack<int> st;
67     Stack<int> st1(20);
68     Stack<int> st2(100);

```

Figure 6 (Stacks of Different sizes)

Activities

Pre –Lab activities:

Task 01: Stack ADT

Stack ADT I

Problem Statement:

In this task, you are required to implement an Abstract Data Type (ADT) for a **fixed-sized stack** using an array. The array should be statically allocated in the stack memory, meaning its size is fixed at compile-time, and it does not grow dynamically. You will need to implement standard stack operations like `push()`, `pop()`, `top()`, and `isEmpty()` while handling edge cases such as overflow and underflow.

Requirements:

- **Fixed Size:** Define a constant size for the stack array.
- **Static Allocation:** The array must be statically allocated, i.e., its memory should not be allocated dynamically (no `new` or heap memory).
- **Operations:**
 - `push(x)`: Adds an element `x` to the top of the stack.
 - `pop()`: Removes the top element from the stack.
 - `Top()`: Returns the top element without removing it.
 - `isEmpty()`: Returns true if the stack is empty, false otherwise.
 - `isFull()`: Returns true if the stack is full, false otherwise.
 - `getCapacity()`: returns the total capacity of the Stack.
 - `getNumOfElems()`: return the number of elements currently in stack.
 - Overloaded Assignment Operator and copy constructor
- **Error Handling:** Handle stack overflow and underflow conditions appropriately.

Example:

- **Test Case: Basic Push and Pop Operations**

```
Stack s;

s.push(10);
s.push(20);
cout << "Top element: " << s.top() << endl; // Expected Output: 20

s.pop();
cout << "Top element after pop: " << s.top() << endl; // Expected
Output: 10

s.pop();
s.pop(); // Should print "Stack Underflow! No element to pop."
```

Stack ADT II

Now modify your above program in such a way that should take size of Stack at run time and allocate memory at heap. For this purpose, Implement dynamic sized stack ADT.

```

3  template <class T>
4  class Stack
5  {
6  private:
7      T* data;
8      int capacity;
9      int top;
10     void resize(int);
11 public:
12     Stack();
13     Stack(const Stack<T>&);
14     Stack& operator= (const Stack<T>&);
15     void push(T);
16     T pop();
17     T stackTop();
18     bool isFull();
19     bool isEmpty();
20     int getCapacity();
21     int getNumberOfElements();
22     ~Stack();
23 };

```

Figure 7 (Dynamic Stack)

Explanation:

At line number 4, a class called "Stack" is defined. In line 6, this class encapsulates private data representing the essential components of a Stack Abstract Data Type (ADT). In line 8, a member variable is introduced, which is a pointer capable of referencing objects of any generic type. Within the public section of the class, various operations associated with the Stack ADT are defined. These member functions are specialized methods tailored to the behaviors expected from a stack ADT.

Example Run:

```

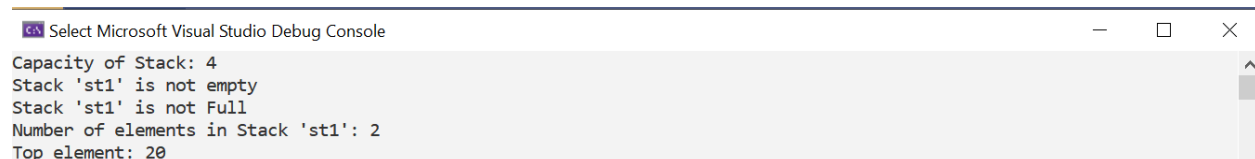
1  #include "Stack.h"
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6      Stack<int> st1, st2;
7      st1.push(10);
8      st1.push(20);
9      st1.push(30);
10     st1.pop(); // this will pop out top most element
11     cout << "Capacity of Stack: " << st1.getCapacity() << endl;
12     if (st1.isEmpty())
13         cout << "Stack \'st1\' is empty" << endl;
14     else
15         cout << "Stack \'st1\' is not empty" << endl;
16
17     if(st1.isFull())
18         cout << "Stack \'st1\' is Full" << endl;
19     else
20         cout << "Stack \'st1\' is not Full" << endl;
21     cout << "Number of elements in Stack \'st1\': " << st1.getNumberOfElements() << endl;
22     cout << "Top element: " << st1.stackTop() << endl;
23     st2 = st1; // Assignment operator
24     Stack<int> st3 = st2; // copy constructor
25     st1.~Stack(); // destructor
26     return 0;

```

Figure 8 (Example run of Stack)

Explanation:

On line 6, we've declared two instances of the Stack class. Between lines 7 and 9, we've pushed three elements onto the stack named st1. On line 10, we've removed the top element from st1 using a pop operation. Subsequently, we've tested the **isFull()** and **isEmpty()** member functions in the following lines. On line 23, the assignment operator is invoked, and on line 22, we've displayed the top element. On line 24, the copy constructor is utilized. Finally, on line 25, the destructor for the st1 instance of the Stack class is called, freeing up the allocated memory.

Output:

```
Select Microsoft Visual Studio Debug Console
Capacity of Stack: 4
Stack 'st1' is not empty
Stack 'st1' is not Full
Number of elements in Stack 'st1': 2
Top element: 20
```

Figure 9 (Output)

Task 03: Balanced Parenthesis

Your task is to implement a function that receives a string and returns true if the parentheses in the expression are balanced otherwise returns false.

You should only consider round brackets “()” as parentheses. Parentheses are considered balanced if each opening parenthesis has its corresponding closing parenthesis and they are properly nested.

For example:

“(a + b) * (c - d)” → balanced

“(((a + b) * (c - d)))” → balanced

“((a + b) * (c - d)” → not balanced (missing closing parenthesis)

“(a + b) * (c - d))” → not balanced (extra closing parenthesis)

bool isBalanced(const string& str);

Think for a while: Does this task really need to be done with stack?

References and Additional Material:

Stack Data Structure

<https://www.geeksforgeeks.org/stack-data-structure/>

<https://www.educative.io/answers/how-to-implement-a-stack-using-an-array-in-cpp>