CC-213L

Data Structures and Algorithms

Laboratory 03

Stack ADT

Version: 1.0.0

Release Date: 05-10-2023

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

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 is s/he 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 followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Madiha Khalid	madiha.khalid @pucit.edu.pk
Teacher Assistants	Muhammad Nabeel	bitf20m009@pucit.edu.pk
	Muhammad Subhan	bcsf20a033@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++:

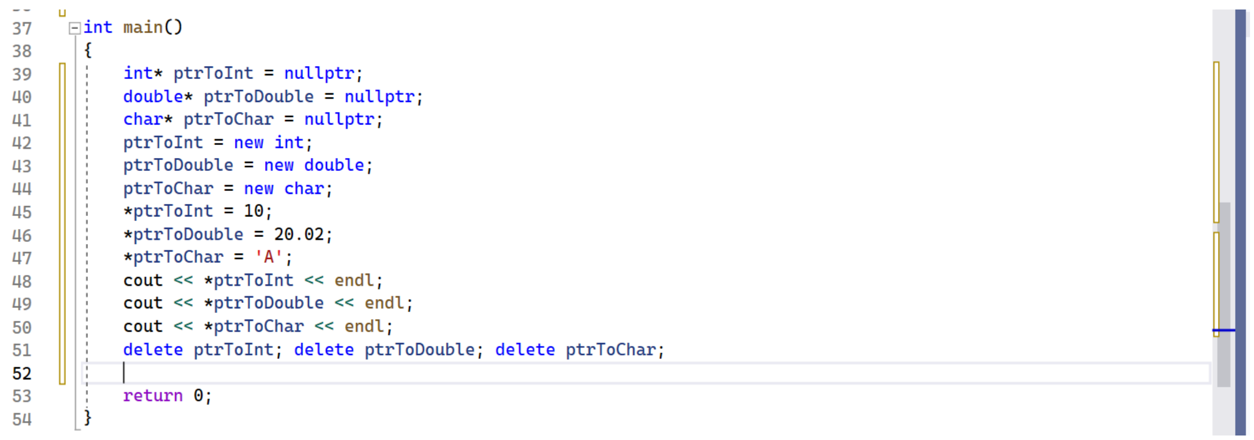
Pointers:

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:



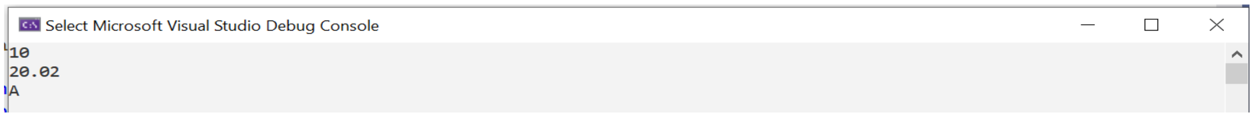
```
37  int main()
38  {
39      int* ptrToInt = nullptr;
40      double* ptrToDouble = nullptr;
41      char* ptrToChar = nullptr;
42      ptrToInt = new int;
43      ptrToDouble = new double;
44      ptrToChar = new char;
45      *ptrToInt = 10;
46      *ptrToDouble = 20.02;
47      *ptrToChar = 'A';
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:



```
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:

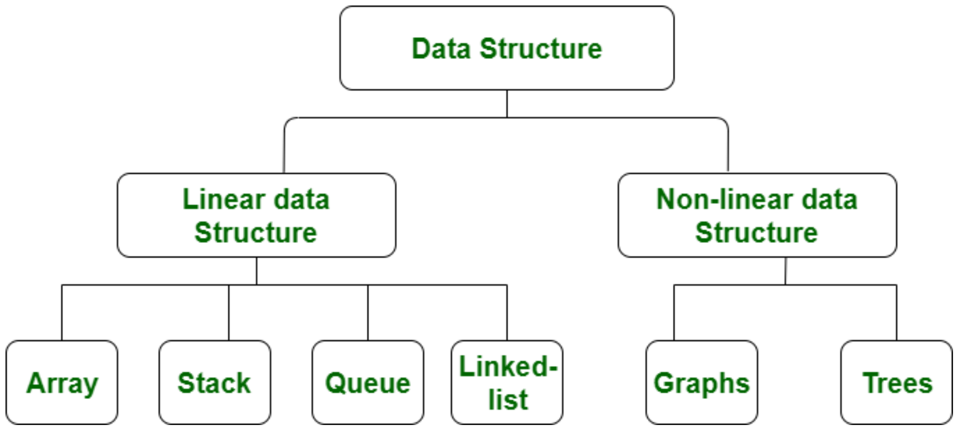


Figure 3 (Types of Data structures)

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:

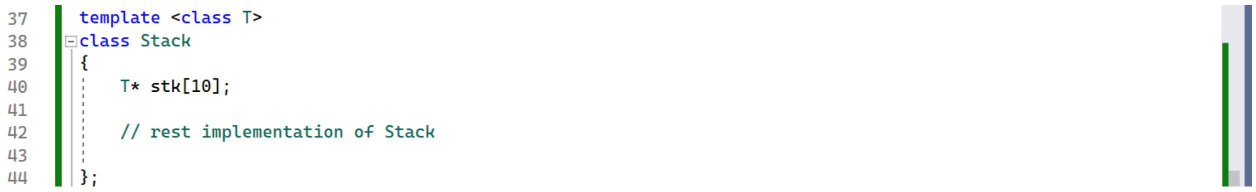
A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It can be visualized as a stack of plates, where the last plate placed on top is the first one to be removed. The two main operations on a stack are push (to add an element to the top) and pop (to remove the top element). 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)

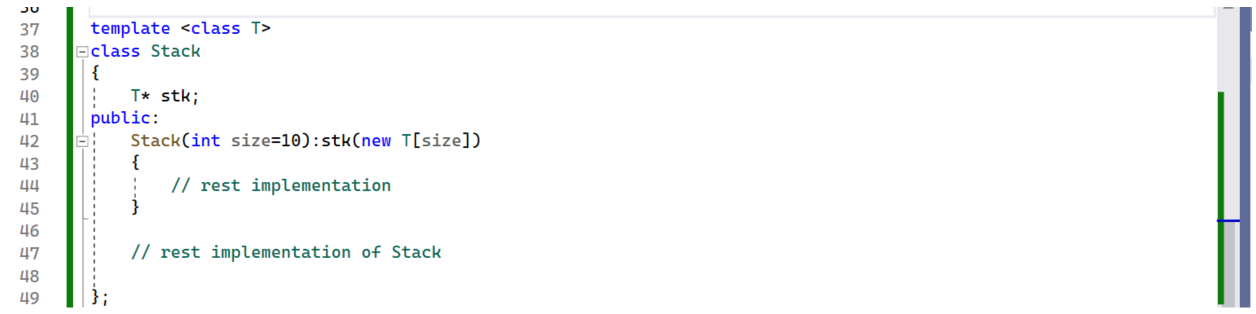
Explanation:

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:

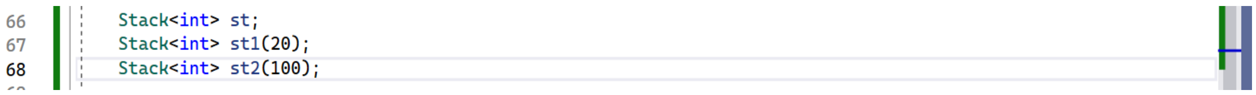


```
37  template <class T>
38  class Stack
39  {
40      T* stk;
41  public:
42      Stack(int size=10):stk(new T[size])
43      {
44          // rest implementation
45      }
46  
47      // rest implementation of Stack
48  
49  };
```

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 20 Stack of 20 size is allocate 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:

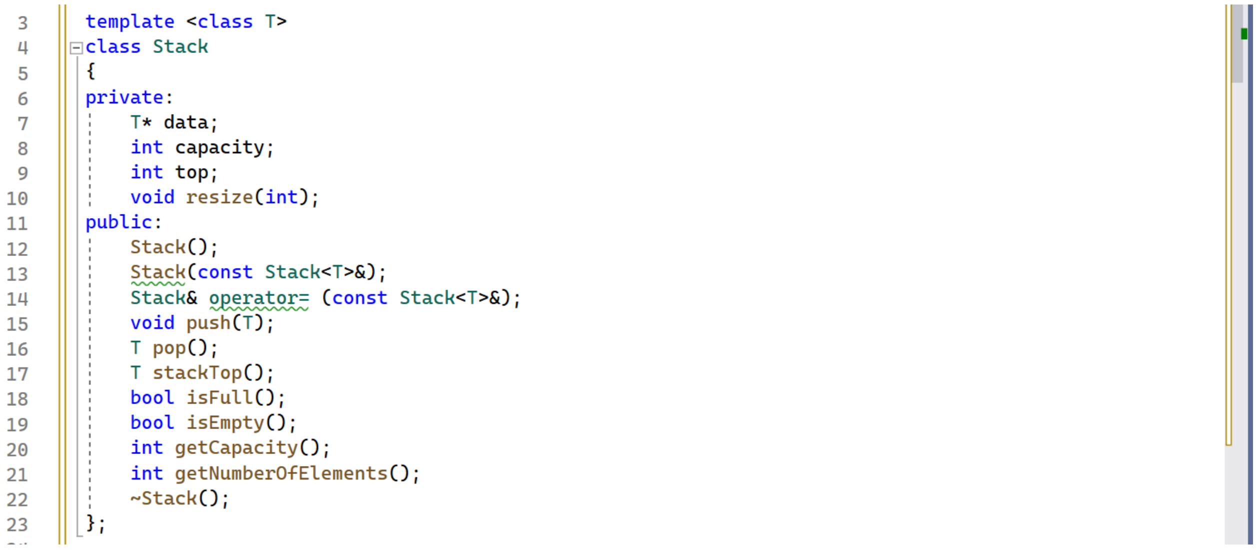
Task 01: Stack ADT

Code a

You have understood in details about the array and memory representation of Array as well as Access and Operation on the Array Class. Keeping in mind all those tasks Implement ADT of fixed sized stack. In this case the size of the Array should be constantly defined at array will be statically allocated at stack memory.

Code b

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.



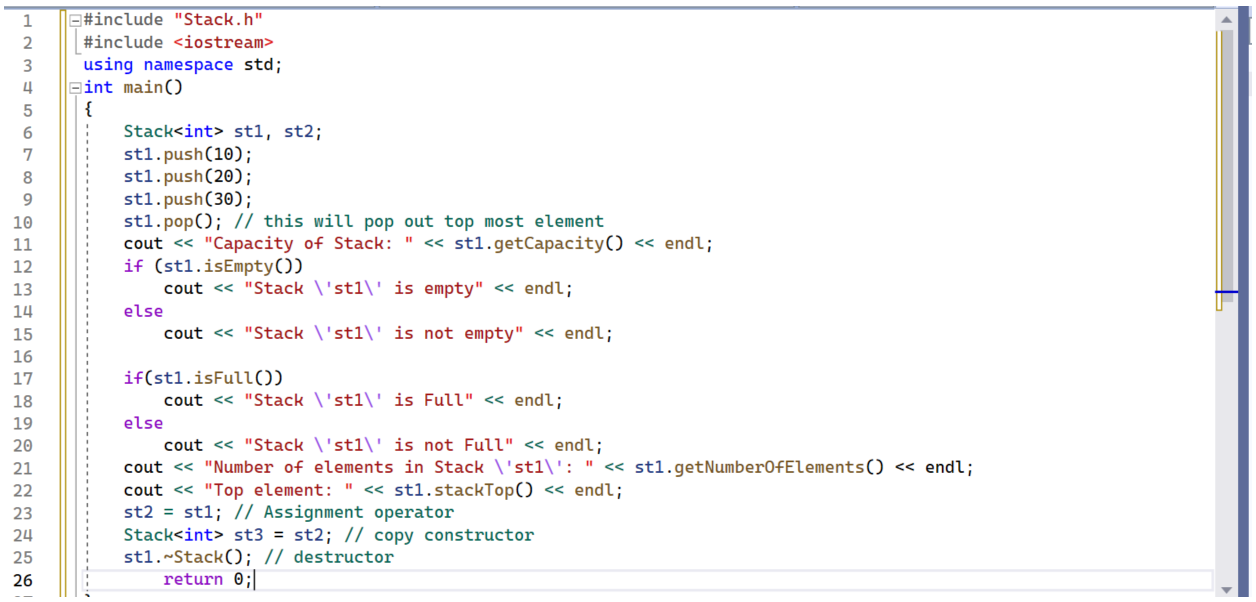
```
3   template <class T>
4   class Stack
5   {
6   private:
7       T* data;
8       int capacity;
9       int top;
10      void resize(int);
11  public:
12      Stack();
13      Stack(const Stack<T>&);
14      Stack& operator= (const Stack<T>&);
15      void push(T);
16      T pop();
17      T stackTop();
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:



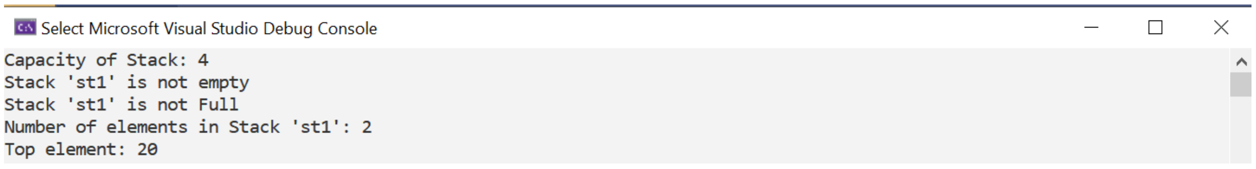
```
1   #include "Stack.h"
2   #include <iostream>
3   using namespace std;
4   int main()
5   {
6       Stack<int> st1, st2;
7       st1.push(10);
8       st1.push(20);
9       st1.push(30);
10      st1.pop(); // this will pop out top most element
11      cout << "Capacity of Stack: " << st1.getCapacity() << 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)

In–Lab Activities:

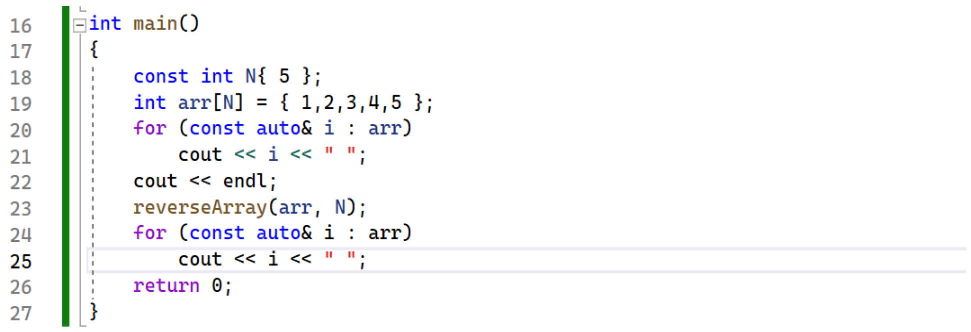
Utilize the implemented Stack Abstract Data Type (ADT) to tackle various programming challenges and solve problems. Apply problem-solving skills by harnessing the power of the stack data structure to address a wide range of programming challenges effectively.

Task 01: Reverse Array

Implement a template function which receives an array and reverses its elements using stack.

void reverseArray(T* arr, int size);

Example runs



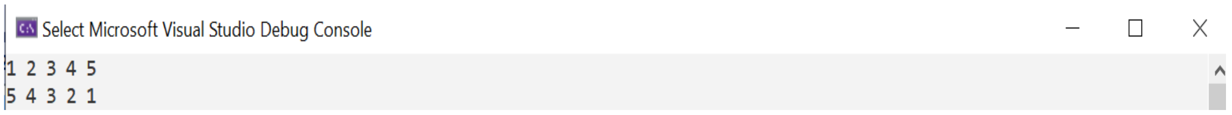
```
16  int main()
17  {
18      const int N{ 5 };
19      int arr[N] = { 1,2,3,4,5 };
20      for (const auto& i : arr)
21          cout << i << " ";
22      cout << endl;
23      reverseArray(arr, N);
24      for (const auto& i : arr)
25          cout << i << " ";
26      return 0;
27  }
```

Figure 10 (Reverse Array)

Explanation:

At line 19, we declare an array with a size of 5, initializing its values. Line 20 is where we display this array. Moving to line 23, the **reverseArray** function is invoked, which takes the array's size and a pointer to the array as parameters. Inside this function, the array's elements are reversed using a stack data structure. Subsequently, on the line following the function call, the array is displayed once more. When this function is executed, it effectively reverses the order of all elements within the array.

Output:



```
1 2 3 4 5
5 4 3 2 1
```

Figure 11 (Output)

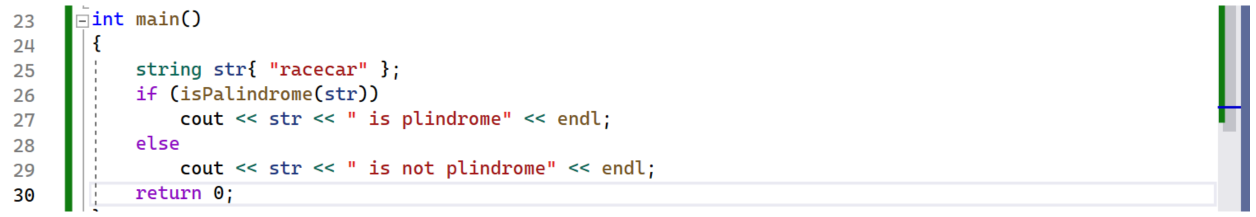
Task 02: String Plindrome

A string is called palindrome if it remains the same even if it is reversed. For example, the string “racecar” is a palindrome because its reversed string is also “racecar”. Your task is to implement a function that receives a string and tells whether it is palindrome or not using stack.

bool isPalindrome(const string& str);

Example Runs

First Example:



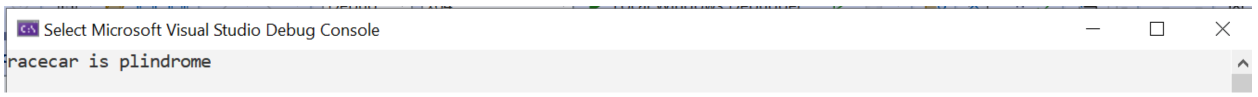
```
23  int main()
24  {
25      string str{ "racecar" };
26      if (isPalindrome(str))
27          cout << str << " is plindrome" << endl;
28      else
29          cout << str << " is not plindrome" << endl;
30      return 0;
```

Figure 12 (string Palindrome)

Explanation:

In line 25 a string named str is initialized with a string that is already a palindrome. If statement header calls isPlidrome function by passing str to it. As string is palindrome so true will be returned and “racecar is palindrome” will be displayed.

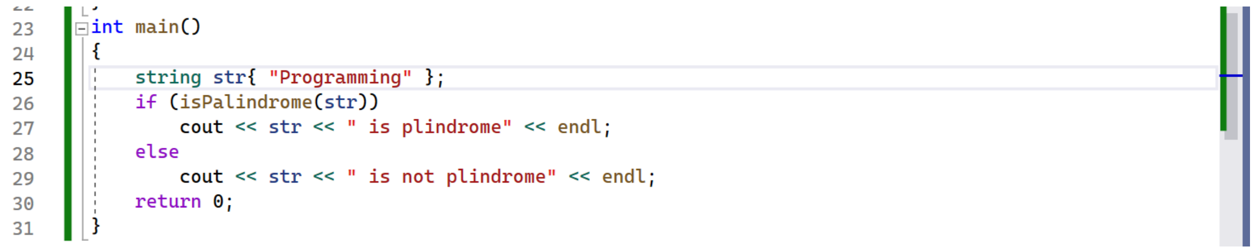
Output:



```
racecar is plindrome
```

Figure 13 (Output)

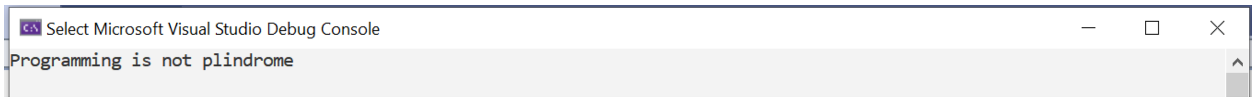
Second Example:



```
23  int main()
24  {
25      string str{ "Programming" };
26      if (isPalindrome(str))
27          cout << str << " is plindrome" << endl;
28      else
29          cout << str << " is not plindrome" << endl;
30      return 0;
31  }
```

Figure 14 (Palindrome)

Output:



```
Programming is not plindrome
```

Figure 15 (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?

Task 04: String words Reverse

Your task is to implement a function that receives a string and reverses each word in it using stack. You can assume that the string only consists of alphabets and spaces. The order of the words should remain same but characters within each word should get reversed.

For example:

String: “Welcome to DSA”

Modified string: “emocleW ot ASD”

void reverseWords(string& str);

Post-Lab Activities

Infix Expression:

An infix expression is a mathematical expression in which operators are placed between operands. In other words, in an infix expression, you write the operator in between the two operands. This is the common way mathematical expressions are typically written in human-readable form. For example:

- 3 + 5
- a * (b + c)
- (2 * x) / (y - 1)

In infix expressions, the precedence of operators determines the order of operations, and parentheses are often used to clarify or change the order of evaluation when needed.

While infix expressions are easy for humans to read and write, they can be more challenging to evaluate programmatically. In contrast, programming languages and computer systems often use postfix (also known as Reverse Polish Notation or RPN) or prefix (also known as Polish Notation) expressions, which are more straightforward to evaluate using a stack-based approach.

To perform calculations with infix expressions in computer programs, they are typically converted to postfix or prefix notation first, and then a stack-based algorithm can be used to evaluate the expression efficiently.

Prefix Expression:

A prefix expression, also known as Polish notation, is a mathematical expression in which the operator precedes its operands. In other words, in a prefix expression, you write the operator before the operands. This notation eliminates the need for parentheses to indicate the order of operations, as it enforces a strict order of evaluation.

For example, the infix expression "3 + 5" in prefix notation would be "+ 3 5." Similarly, the infix expression "(2 * x) / (y - 1)" in prefix notation would be "/ * 2 x - y 1."

To evaluate a prefix expression, you typically use a stack-based algorithm that reads the expression from left to right, pushes operands onto the stack, and when an operator is encountered, it pops the required number of operands from the stack, performs the operation, and pushes the result back onto the stack.

Prefix notation is commonly used in computer science and programming, particularly in languages like Lisp and Forth. It has the advantage of being unambiguous and not requiring parentheses to specify the order of operations.

Postfix Expression:

A postfix expression, also known as Reverse Polish Notation (RPN), is a mathematical expression in which the operator follows its operands. In a postfix expression, you write the operator after the operands. This notation also eliminates the need for parentheses to specify the order of operations because the order of evaluation is determined by the position of the operators.

For example, the infix expression "3 + 5" in postfix notation would be "3 5 +." Similarly, the infix expression "(2 * x) / (y - 1)" in postfix notation would be "2 x * y 1 - /."

To evaluate a postfix expression, you typically use a stack-based algorithm. You read the expression from left to right, pushing operands onto the stack, and when an operator is encountered, you pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.

Postfix notation is commonly used in computer science and programming, particularly in calculators, some programming languages (e.g., Forth), and certain computer architecture instruction sets. It has the advantage of being unambiguous and not requiring parentheses to specify the order of operations.

Pseudo code to convert infix to prefix

```
1.  Reverse infix expression & swap ‘(‘to”)’ & ‘)’ to” (‘
2.  Scan Expression from Left to Right
3.  Print Operands as the arrive
4.  If OPERATOR arrives & Stack is empty, PUSH to stack
5.  IF incoming OPERATOR has HIGHER precedence than the TOP of the Stack, PUSH it on stack
6.  IF incoming OPERATOR has EQUAL precedence with TOP of Stack && incoming
    OPERATOR is ‘^’, POP & PRINT TOP of Stack. Then test the incoming OPERATOR against
    the NEW TOP of stack.
7.  IF incoming OPERATOR has EQUAL precedence with TOP of Stack, PUSH it on Stack.
8.  IF incoming OPERATOR has LOWER precedence than the TOP of the Stack, then POP and
    PRINT the TOP. Then test the incoming OPERATOR against the NEW TOP of stack.
9.  At the end of Expression, POP & PRINT all OPERATORS from the stack
10. IF incoming SYMBOL is ‘(‘PUSH it onto Stack.
11. IF incoming SYMBOL is ‘)’ POP the stack & PRINT Operators till ‘(‘is found or Stack
    Empty.  POP out that ‘(‘from stack
12. IF TOP of stack is ‘(‘PUSH OPERATOR on Stack
13. At the end Reverse output string again.
```

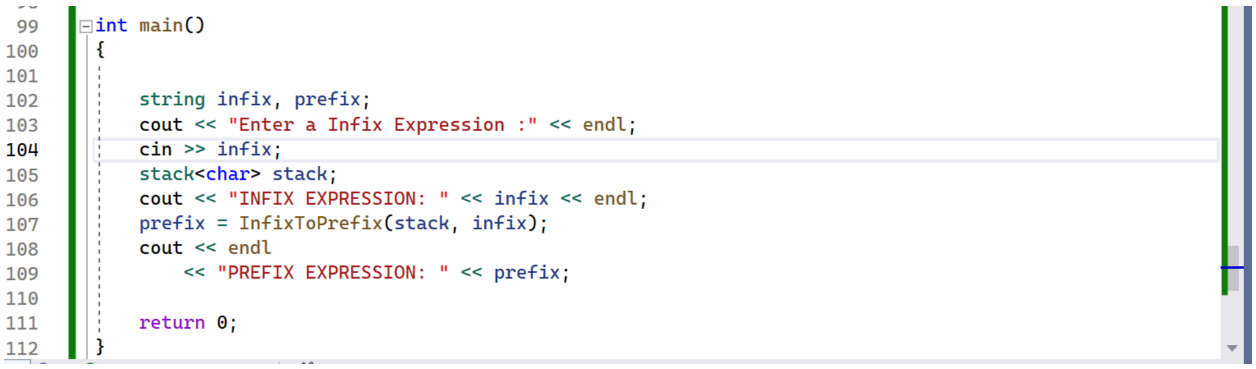
Pseudo code to convert infix to postfix

```
1.  Scan Expression from Left to Right
2.  Print Operands as the arrive
3.  If OPERATOR arrives & Stack is empty, push this operator onto the stack
4.  IF incoming OPERATOR has HIGHER precedence than the TOP of the Stack, push it on stack
5.  IF incoming OPERATOR has LOWER precedence than the TOP of the Stack, then POP and
    print the TOP. Then test the incoming operator against the NEW TOP of stack.
6.  IF incoming OPERATOR has EQUAL precedence with TOP of Stack, use ASSOCIATIVITY
    Rules.
7.  For ASSOCIATIVITY of LEFT to RIGHT –
1.  POP and print the TOP of stack, then push the incoming OPERATOR
8.  For ASSOCIATIVITY of RIGHT to LEFT –
1.  PUSH incoming OPERATOR on stack.
9.  At the end of Expression, POP & print all OPERATORS from the stack
10. IF incoming SYMBOL is ‘(‘PUSH it onto Stack.
11. IF incoming SYMBOL is ‘)’ POP the stack and print Operators till ‘(‘is found. POP that ‘(‘
12. IF TOP of stack is ‘(‘PUSH OPERATOR on Stack
```

Task 01: Program to Convert Infix to Prefix Expression

Implement a C++ program that should take an infix expression and convert to Prefix Expression. Finally display the infix expression as well as Prefix Expression.

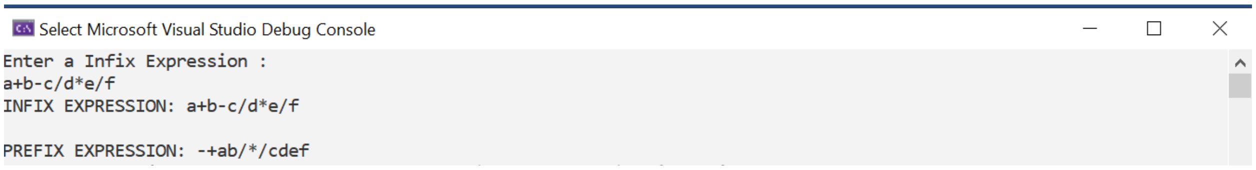
Example Run:



```cpp
int main()
{

    string infix, prefix;
    cout << "Enter a Infix Expression :" << endl;
    cin >> infix;
    stack<char> stack;
    cout << "INFIX EXPRESSION: " << infix << endl;
    prefix = InfixToPrefix(stack, infix);
    cout << endl
        << "PREFIX EXPRESSION: " << prefix;

    return 0;
}
```

Figure 16 (Infix To Prefix)

Output:



Select Microsoft Visual Studio Debug Console

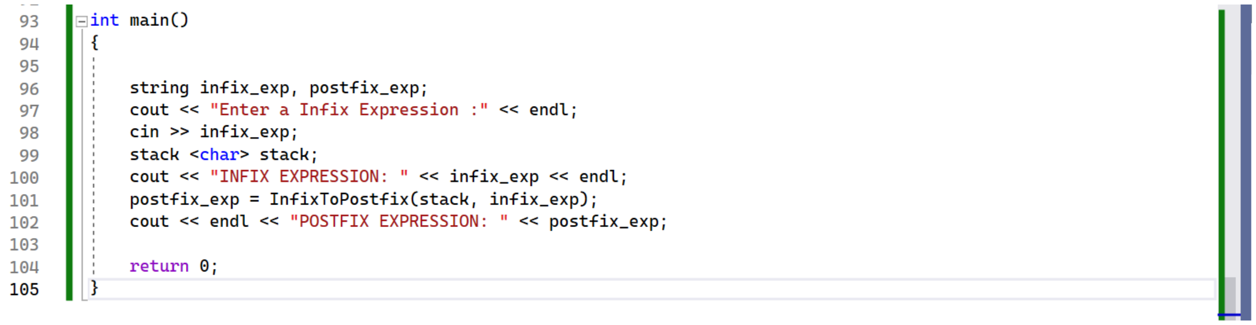
```
Enter a Infix Expression :
a+b-c/d*e/f
INFIX EXPRESSION: a+b-c/d*e/f

PREFIX EXPRESSION: -+ab/*/cdef
```

Figure 17 (Output)

Task 02: Program to Convert Infix to Postfix Expression

Implement a C++ program that should take an infix expression and convert to Postfix Expression. Finally display the infix expression as well as Postfix Expression.

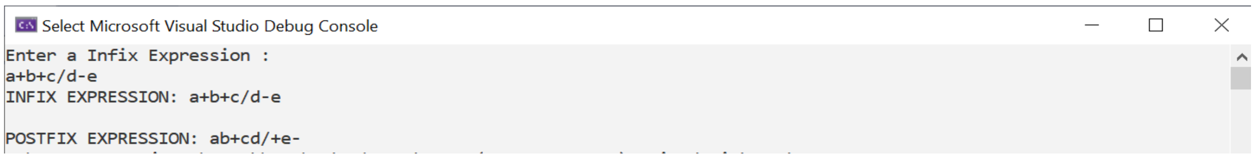
Example Run:



```cpp
int main()
{

    string infix_exp, postfix_exp;
    cout << "Enter a Infix Expression :" << endl;
    cin >> infix_exp;
    stack <char> stack;
    cout << "INFIX EXPRESSION: " << infix_exp << endl;
    postfix_exp = InfixToPostfix(stack, infix_exp);
    cout << endl << "POSTFIX EXPRESSION: " << postfix_exp;

    return 0;
}
```

Figure 18 (Infix to Postfix)

Output:



Select Microsoft Visual Studio Debug Console

```
Enter a Infix Expression :
a+b+c/d-e
INFIX EXPRESSION: a+b+c/d-e

POSTFIX EXPRESSION: ab+cd/+e-
```

Figure 19 (Output)

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA’s will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA’s

Division of Pre-Lab marks:	**[20 marks]**
Task 01: Fixed sized stack	[10 marks]
Task 02: Dynamic Sized stack	[10 marks]
Division of In-Lab marks:	**[30 marks]**
Task 01: Reverse Array	[05 marks]
Task 02: String palindrome	[05 marks]
Task 03: Parenthesis Balance	[10marks]
Task 04: String words reverse	[10marks]
Division of Post-Lab marks:	**[40 marks]**
Task 01: Infix to Prefix Conversion	[20 marks]
Task 02: Infix to Postfix Conversion	[20 marks]

References and Additional Material:

Stack Data Structure

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

Infix to prefix conversion

https://simplesnippets.tech/infix-to-prefix-conversion-using-stack-data-structure-with-c-program-code/

Infix to Postfix conversion

https://simplesnippets.tech/infix-to-postfix-conversion-using-stack-data-structure-with-c-program-code/

Lab Time Activity Simulation Log:

• Slot – 01 – 00:00 – 00:15:	Class Settlement
• Slot – 02 – 00:15 – 00:30:	In-Lab Task 01
• Slot – 03 – 00:30 – 00:45:	In-Lab Task 01
• Slot – 04 – 00:45 – 01:00:	In-Lab Task 02
• Slot – 05 – 01:00 – 01:15:	In-Lab Task 02
• Slot – 06 – 01:15 – 01:30:	In-Lab Task 02
• Slot – 07 – 01:30 – 01:45:	In-Lab Task 03
• Slot – 08 – 01:45 – 02:00:	In-Lab Task 03
• Slot – 09 – 02:00 – 02:15:	In-Lab Task 04
• Slot – 10 – 02:15 – 02:30:	In-Lab Task 04
• Slot – 11 – 02:30 – 02:45:	In-Lab Task 04
• Slot – 12 – 02:45 – 03:00:	Discussion on Post-Lab Task