# KATHMANDU UNIVERSITY

**DHULIKHEL, NEPAL**

**Department of Computer Science & Engineering (DoCSE)**

MINI PROJECT REPORT ON

# “PREFIX TO POSTFIX CONVERSION”

Submitted for the partial fulfillment of

the course COMP-202

Submitted By:

**Yogesh Pant (35)**

**Sambeg Shrestha (45)**

Computer Engineering

2nd Year, 1st Semester

Submitted to:

**Ms. Rajani Chulyadyo**

15 January, 2020

## ABSTRACT

The Mini-Project is one of the partial requirements for the fulfillment of the course COMP-202. Students were required to choose from a list of projects and, thus, we happened to be privileged with performing **a program to convert prefix expressions to postfix, including the time complexity.** We have written a simple C++ that uses the data structure *Stack* to *parse expressions* written in *prefix* notation to *postfix* notation.

We believe practically implementing the theory we have comprehended in the classes into this mini-project has taught us how to work with data structures and how to handle parsing of expressions in notations that a computer understands.

**Keywords:** *C++ Programming, Data Structures, Expression Parsing, Prefix, Postfix, Stack*

## ACKNOWLEDGMENT

We are very grateful to the Department of Computer Science and Engineering, Kathmandu University, especially our lecturer Ms. Rajani Chulyadyo, for providing us a gateway to explore ideas and approaches to implement the learnings of the mandatory coursework COMP-202: the **data structure and algorithms.**

# LIST OF ILLUSTRATIONS

## List of Abbreviations

- ADT – Abstract Data Type
- LIFO – Last In First Out

## List of Figures

- Fig. 1.1 – Difference between different notations
- Fig. 1.2. - Operator precedence and associativity table (highest to lowest)
- Fig. 4.1 – Flowchart of **preToPost** function
- Fig. 4.2- Tabulation for no of steps for cases scenario
- Fig. 4.3 – Test Run

# Table of Contents

ABSTRACT ........ i

ACKNOWLEDGMENT ........ ii

LIST OF ILLUSTRATIONS ........ iii

  List of Abbreviations ........ iii

  List of Figures ........ iii

Chapter 1: Introduction ........ 1

  1.1 Background: ........ 1

  1.2 Objectives: ........ 5

    Primary Objectives: ........ 5

    Secondary Objectives: ........ 5

  1.3 Motivation and Significance: ........ 5

Chapter 2: Related Works ........ 6

Chapter 3: Design and Implementation ........ 7

  3.1. System Requirement Specification ........ 7

    3.1.1. Software Specification ........ 7

Chapter 4: Discussion on the Achievements ........ 8

Chapter 5: Conclusion and Recommendation ........ 17

References ........ 18

Bibliography ........ 18

# Chapter 1: Introduction

## 1.1 Background:

The way of writing arithmetic expression is called a notation. An arithmetic expression consists of operators and operands—the operators, +, -, etc., perform arithmetic tasks on the operands, constant literals like integers (*-1, 40, 0, etc*.) or well-defined variables (*A, B, etc*.). These expressions can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are:

1. Infix Notation
2. Prefix (Polish) Notation
3. Postfix (Reverse-Polish) Notation

These notations are named depending on their usage of the operators used in the expression. Following is an overview of the different notations:

1. **Infix Notation:**

   Infix notation is usually how humans write expressions, e.g. ***a* - *b* + *c***, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

2. **Prefix Notation:**

   As the name implies, in this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, **+*ab***. This is equivalent to its infix notation ***a* + *b***. Prefix notation is also known as **Polish Notation**.

### 3. Postfix Notation

Contrasting to Prefix, in this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ***ab+***. This is equivalent to its infix notation ***a + b***. This notation style is also known as **Reversed Polish Notation.**

| S.N | Infix Notation | Prefix Notation | Postfix Notation |
|---|---|---|---|
| 1 | a + b | + a b | a b + |
| 2 | (a + b) ∗ c | ∗ + a b c | a b + c ∗ |
| 3 | a ∗ (b + c) | ∗ a + b c | a b c + ∗ |
| 4 | a / b + c / d | + / a b / c d | a b / c d / + |
| 5 | (a + b) ∗ (c + d) | ∗ + a b + c d | a b + c d + ∗ |
| 6 | ((a + b) ∗ c) - d | - ∗ + a b c d | a b + c ∗ d - |

Fig. 1.1 – Difference between different notations

**Parsing Expressions:**

It is not efficient to create an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

**Precedence:**

In cases of multiple operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example − ***a + b * c → a + ( b * c )***

As multiplication operation has precedence over addition, ***b * c*** will be evaluated first.

**Associativity:**

In cases of operators with the same precedence, we follow the associativity rule. For example, in the expression: ***a* + *b* − *c***, both + and – have the same precedence, then the part of the expression to be evaluated first is determined by associativity of those operators. Here, both + and − are left-associative, so the expression is evaluated as ***(a* + *b)* − *c***.

Precedence and associativity determine the order of evaluation of an expression.

| S.N. | Operator | Precedence | Associativity |
|---|---|---|---|
| 1 | Exponentiation ^ | Highest | Right Associative |
| 2 | Multiplication ( ∗ ) & Division ( / ) | Second Highest | Left Associative |
| 3 | Addition ( + ) & Subtraction ( − ) | Lowest | Left Associative |

Fig. 1.2. - Operator precedence and associativity table (highest to lowest)

The above table shows the default behavior of operators. At any point in time in expression evaluation, the order can be altered by using parenthesis. For example −

In ***a* + *b* * *c***, the expression part ***b* * *c*** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for ***a* + *b*** to be evaluated first, like ***(a* + *b)* * *c***.

To implement conversion from one expression to the other, we implement Data Structures.

## Data Structure and Algorithms - Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

## Stack Representation

A stack can be implemented using Array, Structure, Pointer, and Linked List. A stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

## Basic Operations

Stack operations may involve initializing the stack, using it, and then de-initializing it. Apart from this basic stuff, a stack is used for the following two primary operations −

**push()** − Pushing (storing) an element on the stack.

**pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

**peek()** − get the top data element of the stack without removing it.

**isFull()** − check if stack is full.

**isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. This pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.

## 1.2 Objectives:

### Primary Objectives:

- To implement data structure in practical programming
- To learn parsing of expressions from one form to other

### Secondary Objectives:

- To learn coding and programming language

## 1.3 Motivation and Significance:

As an aspiring computer science and engineering students, it is very necessary to know how a computer functions on both hardware and software levels. Knowing and understanding fundamentals like data structures as well as inner working of expression notations we can get acquainted with the broader world of computing technology.

# Chapter 2: Related Works

Data Structure and Algorithm is a very fundamental topic rudimentary for any computer software design. Information and knowledge on various aspects of data structures, their functions, their applications and their necessity are prevailing over the internet innumerably and are well resourced in books as well.

Particularly for this mini-project, articles regarding data structures and algorithms related to conversion of prefix to postfix expressions were studied. The following sources were consolidated in our research and referenced to complete our project.

1. **GeeksforGeeks:**

   A Computer Science portal for **geeks**. It contains well written, well thought, and well-explained computer science and programming articles, quizzes and many more.

2. **TutorialsPoint:**

   Tutorials Point is an online portal that consists of tutorials that flaunts a wealth of tutorials and allied articles on topics ranging from programming languages to web designing to academics and much more.

# Chapter 3: Design and Implementation

## 3.1. System Requirement Specification

### 3.1.1. Software Specification

Running the code requires a C++ compiler. There are different compilers available in different platforms.

**Windows:**

You can download the latest MinGW software for windows_based devices for compiling, debugging, and creating executable windows programs from the following link:

https://sourceforge.net/projects/mingw-w64/

**Linux:**

Follow the following tutorial to run C++ programs on Linux:

https://www.cyberciti.biz/faq/howto-compile-and-run-c-cplusplus-code-in-linux/

**macOS:**

Follow the following video tutorial to set-up C++ on Mac devices:
https://www.youtube.com/watch?v=1E_kBSka_ec

You can also use online resources to run C++ code.

**Web-based Compiler:**

https://www.onlinegdb.com/online_c++_compiler

# Chapter 4: Discussion on the Achievements

The working file directory of the code is illustrated in the tree below:

**File structure tree**:

```
├──────── pretopost.h
└──────── pretopost.cpp
```

The snippets to our working code  included as follows:

**pretopost.h**

```
#include <string>


using namespace std;
#define MAX_STACK_SIZE 100


class stack
{
public:
    stack();
    ~stack();


    void push(string ele);
    string pop();
    string topc();
    bool isEmpty();
    bool isFull();
    // char preToPost(char pre_exp);


    string elements[MAX_STACK_SIZE];
    char top;
};
```

## pretopost.cpp

```
// Program to convert prefix expressions to postfix expressions
#include <iostream>
#include <string>
#include "pretopost.h" // includes the header file.
using namespace std;

//definitions of constructors and destructors
stack::stack(){                                             //(1)
    top = -1;
}

stack::~stack(){}

// definition of isEmpty function
bool stack::isEmpty(){                                      //(2)
    if (top < 0) return true;
    else return false;
}

// definition of isFull function
bool stack::isFull(){                                       //(2)
    if (top >= (MAX_STACK_SIZE - 1)) return true;
    else return false;
}

// definition of push function
void stack::push(string ele){                               //(3)
    if (isFull()){
        cout << " cannot push " << ele << " \n Stack overflow";
    }
    else{
        top++;
        //cout << " \n pushed \n ";
        this->elements[top] = ele;
    }
}

// definition of pop function
string stack::pop(){                                        //(2)
    if (top < 0){
        cout << " cannot pop \n stack underflow ";
    }
    else{
        //cout << "\n popped \n";
        return this->elements[top--];
    }
}
```

```
// definition of topc function
string stack::topc(){                               //(2)
    if (isEmpty()) cout << "Stack underflow \n";
    else return this->elements[top];
}

// funtion to check if the character is an operator or not
bool isOperator(char x){
    switch (x){
    case '+':
    case '-':
    case '/':
    case '*':
        return true;
    }
    return false;
}

// the conversion happens here
string preToPost(string pre_exp){
    stack s;

    // getting the length of expression
    int length = pre_exp.size();

    // reading the expression from right to left
    for (int i = length - 1; i >= 0; i--){                //(length)
        // checking if the symbol is an operator
        if (isOperator(pre_exp[i])){
            // pops two operands from the stack
            string op1 = s.topc();
             s.pop();
            string op2 = s.topc();
             s.pop();

            // concatenate the operands and the operator
            string temp = op1 + op2 + pre_exp[i];         //((length-1)/2)

            // Pushing the concatenated string back to stack
            s.push(temp);
        }

        // if the symbol is an operand
        else{
            // pushing the operand to the stack
            s.push(string(1, pre_exp[i]));                //( (length+1)/2 )

        }
    }

    // returning the Postfix expression
    return s.topc();
}
```

```
// MAIN DRIVER CODE
int main()
{
    string pre_exp;
    cout << "\n ENTER ANY PREFIX EXPRESSION \n ";          //(1)
    cin >> pre_exp;
    cout << "Postfix : " << preToPost(pre_exp);            //(1)
    return 0;
}
```

Github: https://github.com/sthasam2/CE2018_MP_35_45

**Rundown of the code:**

***pretopost.h :***

*'pretopost.h'* is the header file of the program which consists of all the declarations of functions and variables to be used in the program, mainly this file includes components of the Stack data.

***pretopost.cpp :***

*'pretopost.cpp'* is the source file of the programs consisting of the data structure function definitions, pre to post conversion function definition, and the driver module.

A brief summary of all the functions:

| | |
|---|---|
| Lines 1-5: | **include** block + **using** namespace |
| Lines 7-10: | stack *constructor*- definition for initializing stack class |
| Line 12: | stack *destructor*- definition for destructing class |
| Lines 14-18: | isEmpty(): bool definition for returning if stack is emplty or not |
| Lines 20-24: | isFull(): bool definition for returning if stack is full or not |
| Lines 26-36: | push(): definition for pushing an element into stack |
| Lines 38-47: | pop(): definition for popping an element from stack |
| Lines 48-52: | topc(): definition for returning the topmost element of stack |
| Lines 54-64: | isOperator (): definition to check given character is operator(+,-,*,/) or not |

Lines 66-99: preToPost( (): definition for converting a given prefix expresson to postfix

Lines 101-108: main(): the main driver function

**Algorithm for Prefix to Postfix ( preToPost() ):**

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack

  Create a string by concatenating the two operands and the operator after them.

  string = operand1 + operand2 + operator

  And push the resultant string back to Stack
- Repeat the above steps until the end of Prefix expression.

**Flowchart:**

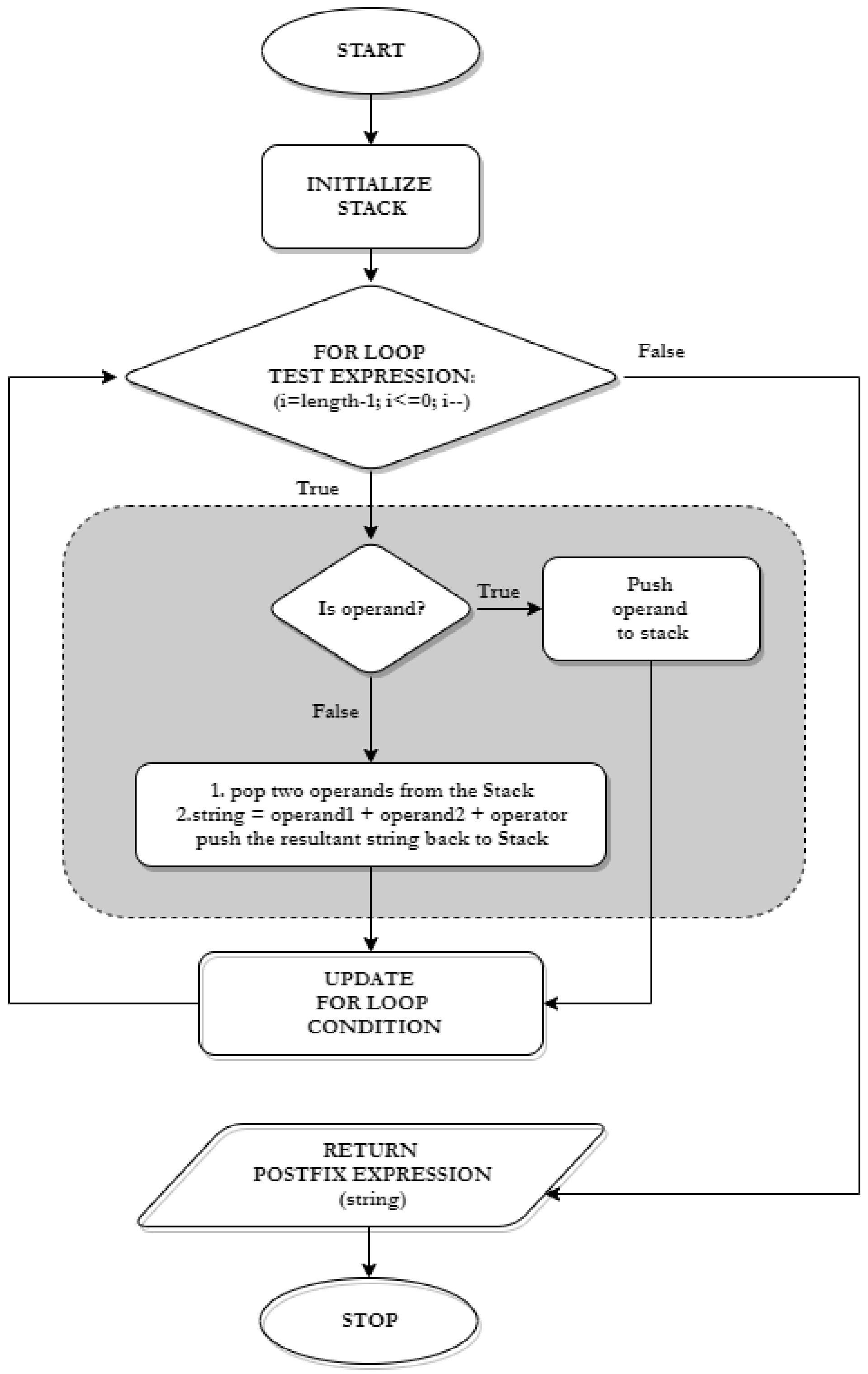

START
INITIALIZE STACK
FOR LOOP TEST EXPRESSION: (i=length-1; i<=0; i--)
False
True
Is operand?
True
Push operand to stack
False
1. pop two operands from the Stack
2.string = operand1 + operand2 + operator
push the resultant string back to Stack
UPDATE FOR LOOP CONDITION
RETURN POSTFIX EXPRESSION (string)
STOP

Fig. 4.1 – Flowchart of **preToPost** function

## Time Complexity:

For the time complexity of the program, we look at the core function *preToPost(),* which converts the prefix expression to postfix expression. Other functions in the *pretopost.cpp* are either data structure functions or the bool function to determine operands whose complexity does not exceed the worst-case scenario (The number of steps for the functions is commented in the snippet).

So, we take the ***preToPost(string pre exp)*** function to measure the complexity of the program. The different case scenarios for the function are tabulated below:

| S.N. | Code | Best Case | Worst Case |
|---|---|---|---|
| 1 | `string preToPost(string pre_exp){` | 0 | 0 |
| 2 | `    stack s;` | 0 | 0 |
| 3 | `    int length = pre_exp.size();` | 0 | 0 |
| 4 | `    for (int i = length - 1; i >= 0; i--){` | 3 | length |
| 5 | `        if (isOperator(pre_exp[i])){` | 1 | (length-1)/2 |
| 6 | `            string op1 = s.topc();` | 1 | (length-1)/2 |
| 7 | `            s.pop();` | 1 | (length-1)/2 |
| 8 | `            string op2 = s.topc();` | 1 | (length-1)/2 |
| 9 | `            s.pop();` | 1 | (length-1)/2 |
| 10 | `            string temp = op1 + op2 + pre_exp[i];` | 1 | (length-1)/2 |
| 11 | `            s.push(temp);` | 1 | (length-1)/2 |
| 12 | `        }` | 0 | 0 |
| 13 | `        else{` | 2 | (length+1)/2 |
| 14 | `            s.push(string(1, pre_exp[i]));` | 2 | (length+1)/2 |
| 15 | `        }` | 0 | 0 |
| 16 | `    }` | 0 | 0 |
| 17 | `    return s.topc();` | 1 | 1 |
| 18 | `}` | 0 | 0 |
|  | **Total** | 15 | 9*Length-3/2 |

Fig. 4.2- Tabulation for no of steps for cases scenario

Therefore, if length=n, ***Maximum: O(n)***

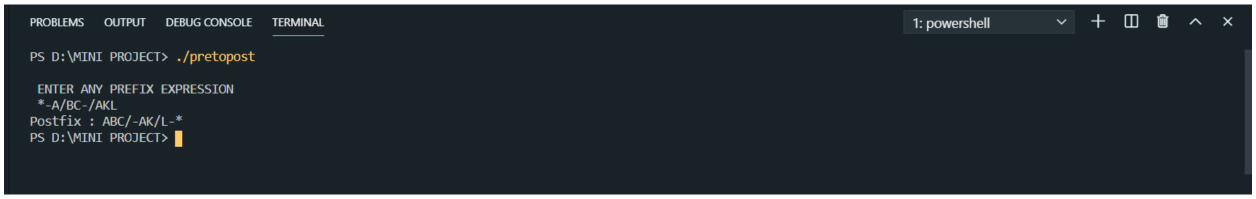
**Test run of the project:**

To test the program, first the code was compiled then the executable was run.

A sample prefix expression: ***-A/BC-/AKL**

was entered according to the instructions of the program.

The following output was generated:



```
PS D:\MINI PROJECT> ./pretopost

 ENTER ANY PREFIX EXPRESSION
 *-A/BC-/AKL
Postfix : ABC/-AK/L-*
PS D:\MINI PROJECT>
```

Fig. 4.3 – Test Run

The corresponding postfix expression generated was:  ***ABC/-AK/L-***

# Chapter 5: Conclusion and Recommendation

The mini-project is an excellent way of putting academics into a practical use case. Learning and implementing said theories practically in a real-world scenario helps in better understanding the content and course matter. Our project that entailed the conversion of prefix expressions to postfix expressions was a step towards utilizing the knowledge of data structure meanwhile understanding how a computer parses expression was excellent academic practice for learning.

Adding more functionality to the project, like detecting errors, exceptions, and unnecessary garbage input can be done further to consolidate the program.

## References

Data Structure - Expression Parsing - Tutorialspoint. (n.d.). Retrieved January 14, 2020, from https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm

Data Structure and Algorithms - Stack - Tutorialspoint. (n.d.). Retrieved January 14, 2020, from https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

Prefix to Postfix Conversion. (2019, May 15). Retrieved January 14, 2020, from https://www.geeksforgeeks.org/prefix-postfix-conversion/

## Bibliography

E. Balgurusamy. (2008), Object Oriented Programming with C++ Fourth Edition.