# A Review of different algorithms for Infix to Postfix Conversion

**Md. Saifur Rahman**
2018-1-80-048
CSE Dept.East West University, Dhaka.Bangladesh
**Email:**2018-1-60-048@std.ewubd.edu

**Taiammum Uday**
2018-1-80-046
CSE Dept.East West University, Dhaka.Bangladesh
**Email:**2018-1-60-046@std.ewubd.edu

**Obaida Jahan**
2019-2-80-080
CSE Dept.East West University, Dhaka.Bangladesh
**Email:** 2019-2-60-080@std.ewubd.edu

## Abstract

**In Computer Science, Reverse Polish notation has made the calculation simple and has benefitted the new technologies by minimizing computational time. RPN is being implemented in calculators since 1960 because of its easy and simple implementation but high-performance. In this paper, we have reviewed some approaches for infix to postfix expressions conversion by following some rules and have highlighted some of the applications and advantages of existing methods. The algorithms we have reviewed here are driven from other existing algorithms through their human readability. We have discussed a comparative analysis of RPN, Shunting Yard algorithm and another approach to Post fix conversion named as PKR algorithm.**

**Keywords–Infix Notation, Postfix Notation, Data Structure, Keystrokes.**

## I. INTRODUCTION

Reverse Polish algorithm is being used to represent the expressions where the operator is placed after the operands/arguments. Jan Lukasiewicz [1] invented in 1920.

Second method is Shunting Yard algorithm which was first introduced by Edsger Dijkstra. Shunting-Yard algorithm does the mathematical expressions on specified infix expressions. It produces output in both RPN and AST. This algorithm performs its operations on stack and queue where stack holds operators in it and output is added into the Queue. Operator precedence parsing [2] is the generalized form of shunting-yard algorithm.

## II. INFIX TO POST-FIX NOTATION

**Infix notation** is an arithmetic and logical notation- represents the operator placed between two operands. It is difficult for computer to parse infix notation. The order of operations is mandatory to indicate and operands and operators must be surrounded by parentheses in infix notation [3].

**Post-fix notation** is a mathematical notation which-used to parse a machine. Post-fix notation is also known as Reverse Polish Notation (RPN) and involves operands followed by operators. Use of parentheses is not necessary for calculations in RPN[1].

## III. WHY CONVERSION IS NEEDED FROM INFIX TO POST-FIX NOTATION?

RPN has the property that brackets are not required to represent the grouping of the terms or order of evaluation unlike Infix expression[4]. By push-pop operation on STACK, postfix expression can easily be obtained. This greatly simplifies the expression's computation within computer programs [5]. The big advantage of RPN is that for a computer to analyze notations, it is `extremely easy and fast. Infix notation is easier to read for humans, but a machine can easily parse postfix notation. In postfix expression we can obtain the original parse tree without original knowledge, but the same is not applied to infix expressions.

## IV. HISTORICAL IMPORTANCE AND APPLICATION AREAS

### *A. History of Implementations*

Hewlett-Packard Engineers designed first calculator- used RPN, HP9100A in 1968- which was regarded as Desktop Calculator [6]. Second is HP-35, which is the first handheld scientific calculator in world, used RPN in 1972 [7]. HP introduced LCD-based calculator in the 1980s such as HP 10C, HP 11C, HP 15C, HP 16C and HP 12C which is famous financial calculator. From 1990 to 2003, RPN calculators with graphing includes HP-48 series and in 2006 HP-50g manufactured. In 2011, HP manufactured 12C, 12C platinum, 17 BII, 20 B (financial), 30 B (business), 33S, 35S, 48 GII and 50G (scientific) [8].

Prinztronic brand developed PROGRAM which was Programmable Scientific Calculator. Soviet programmable calculators such as MK-52, MK-61, B3-34 and earlier B3-21, used RPN for both automatic mode and programming[9].

### ***B. Current Implementations***

Exiting implementations using RPN include:

- Stack oriented programming languages like Forth, Factor, and PostScript page description.

- Hardware calculators include HP science/ engineering, business/ finance and Semico calculators.
- Software calculators include Mac OS X calculator, Apple's "reverse polish notation calculator", Android's "RealCalc", UNIX system calculator program dc, etc.

## V. EXISTING ALGORITHMS TO CONVERT INFIX EXPRESSIONS INTO POSTFIX EXPRESSIONS

(a). Reverse Polish Notation Algorithm[1]:

**RPN (I, P)**

Suppose, I is the Infix notation arithmetic expression. This algorithm gives Postfix expression P.

1. Push "("onto Stack and add ")" to the end of I.
2. Scan the expression I from Left to Right and repeat from step number 3 to 6 for every element of I until stack is empty:
3. If an operand is read, add it to P.
4. If a left parenthesis is read, push onto Stack.
5. If an operator, say $, is read, then :

(a) Repeatedly, each operator is added to P by popping from Stack which has the same/higher precedence than the operator encountered $.

(b) Add $ to Stack.

6. If a right parenthesis ")" is read, then:

(a) Repeatedly, each operator is added to P by popping from Stack until a left parenthesis "("is read.

(b) Remove left parentheses "(". [Do not add it to P].

7. Exit.

**Numerical Example:**

I: A + (B * C - (D / E ^ F) * G) * H

Step 1: A + (B * C - (D / E ^ F) * G) * H, read expression from left to right.

TABLE I. CONVERSION OF INFIX TO POSTFIX EXPRESSION BY USING RPN ALGORITHM.

| Symbol Scanned | | STACK | Expression P |
|---|---|---|---|
| **(1)** | **A** | **(** | **A** |
| **(2)** | **+** | **( +** | **A** |
| **(3)** | **(** | **( + (** | **A** |
| **(4)** | **B** | **( + (** | **AB** |
| **(5)** | ***** | **( + ( *** | **AB** |
| **(6)** | **C** | **( + ( *** | **ABC** |
| **(7)** | **–** | **( + ( -** | **ABC*** |
| **(8)** | **(** | **(+ ( - (** | **ABC*** |
| **(9)** | **D** | **(+ ( - (** | **ABC*D** |
| **(10)** | **/** | **(+ ( - ( /** | **ABC*D** |
| **(11)** | **E** | **(+ ( - ( /** | **ABC*DE** |
| **(12)** | **^** | **(+ ( - ( / ^** | **ABC*DE** |
| **(13)** | **F** | **(+ ( - ( / ^** | **ABC*DEF** |
| **(14)** | **)** | **(+ ( -** | **ABC*DEF^/** |
| **(15)** | ***** | **(+ ( - *** | **ABC*DEF^/** |
| **(16)** | **G** | **(+ ( - *** | **ABC*DEF^/G** |
| **(17)** | **)** | **(+** | **ABC*DEF^/G*-** |
| **(18)** | ***** | **(+ *** | **ABC*DEF^/G*-** |
| **(19)** | **H** | **(+ *** | **ABC*DEF^/G*-H** |
| **(20)** | **)** | **EMPTY** | **ABC*DEF^/G*-H*+** |

(b)Shunting-yard algorithm [2].

**Algorithm**

Postfix expression is added to output QUEUE. Consider there is an Infix expression then the algorithm says that: Read a token.

1.1. If it is an operand, then add it to the Queue.
1.2. If it is an operator, then push it onto the Stack.
1.3. If it is an argument separator (e.g., a comma):
  (a) Then add operators to Queue from Stack until a left parenthesis is at the top of Stack. If no left parenthesis is there, either parentheses were mismatched or separator was misplaced.
1.4. If it is an operator, $, then:
  (a) while there is an operator, #, at the top of the Stack, and
    1.4.a.1. either $ is left-associative and its precedence is less than or equal to that of #, or $ has low precedence than that of #, then
      1.4.a.1.1. pop # off the Stack, onto the output Queue;
      1.4.a.1.2. Push $ onto the Stack.
1.5. If it is a "("left parenthesis, then push it to Stack.
1.6. If it is a ")"right parenthesis:

(a) Pop operators from Stack to Queue until left parenthesis is read at the top of Stack.
(b) The left parenthesis is popped but not added to the Queue.
(c) If the token is operator, pop it to Queue.
(d) If no left parenthesis is inside the top of Stack, then there are mismatched parentheses.

1.7. When no other token left to read:
  (a) While there are operators in the Stack
    1.7.a.1. If operator is a parenthesis, then mismatched parentheses error is there.
    1.7.a.2. Pop the operator onto Queue.
1.8. Exit.

**Numerical Example on Shunting-Yard Algorithm**

Consider the Infix Expression: A + (B * C - (D / E ^ F) * G) * H

Transform given in-fix into its post-fix expression in OUTPUT QUEUE.

Step 1: A + (B * C - (D / E ^ F) * G) * H, read expression from left to right.

TABLE II. CONVERSION OF INFIX TO POSTFIX EXPRESSION BY USING SHUNTING-YARD ALGORITHM.

| Token | | Action | Output (in RPN) | Operator Stack | Notes |
|---|---|---|---|---|---|
| **(1)** | A | Add A to output | A | | |
| **(2)** | + | Push + to stack | A | + | |
| **(3)** | ( | Push ( to stack | A | + ( | |
| **(4)** | B | Add B to output | AB | + ( | |
| **(5)** | * | Push * to stack | AB | + ( * | |
| **(6)** | C | Add C to output | ABC | + ( * | |
| **(7)** | – | Pop stack to output<br>Push - to stack | ABC* | + ( - | - has less precedence than * |
| **(8)** | ( | Push ( to stack | ABC* | + ( - ( | |
| **(9)** | D | Add D to output | ABC*D | + ( - ( | |
| **(10)** | / | Push / to stack | ABC*D | + ( - ( / | |
| **(11)** | E | Add E to output | ABC*DE | + ( - ( / | |
| **(12)** | ^ | Push ^ to stack | ABC*DE | + ( - ( / ^ | |
| **(13)** | F | Add F to output | ABC*DEF | + ( - ( / ^ | |
| **(14)** | ) | Pop stack to output<br><br>Pop stack | ABC*DEF^/<br><br>ABC*DEF^/ | + ( - (<br><br>+ ( - | Repeated until "(" found<br>Discard matching parenthesis |
| **(15)** | * | Push * to stack | ABC*DEF^/ | + ( - * | * has high precedence than - |
| **(16)** | G | Push G to stack | ABC*DEF^/G | + ( - * | |
| **(17)** | ) | Pop ) to output<br><br>Pop stack | ABC*DEF^/G*-<br><br>ABC*DEF^/G*- | + (<br><br>+ | Repeated until "(" found<br>Discard matching parenthesis |
| **(18)** | * | Push * to stack | ABC*DEF^/G*- | + * | |
| **(19)** | H | Add H to output | ABC*DEF^/G*-H | + * | |
| **(20)** | *End* | Pop entire stack and add to output | ABC*DEF^/G*-H*+ | | |

## VI. PKR APPROACH FOR TRANSFORMING INFIX TO POSTFIX EXPRESSIONS

PKR algorithm uses either of the two rules to compare two operators**:**

1. **1.** BEMD%AS refers to Bracket Exponent Multiply Divide Modulus Addition Subtract

   ← Yes if order of operator results in Yes then operator can stay inside stack

   No → if order of operator results in No then operator which is pushed first, is popped and added to output expression.

2. **2.** BNAO refers to Bracket NOT (Binary) AND (Binary) OR (Binary)

   ← Yes if order of operator results in Yes, the operator can stay inside stack

   No → if order of operator results in No, the operator which is pushed first, is popped and added to output expression.

**Algorithm:**

Let P is Infix arithmetic expression. This algorithm will give the equivalent Postfix expression in OUTPUT.

1. SCAN P from Left to Right, the encountered symbol is pushed onto STACK and if
   (a) Operand encountered, add it to OUTPUT.
   (b) Operator encountered, push onto STACK.
   (c) If open parenthesis encountered "(", push onto STACK.
2. If closed parenthesis ")" encountered then pop operators in LIFO order and add it to OUTPUT and it will also cancel the "(" opening parenthesis.

3. If two operators encountered continuously the follow BEMD%AS rule -
   (a) If sequence of operators is OCCURING from Right to Left in BEMD%AS then no popping.
   (b) If sequence of operators is OCCURING from Left to Right then pop the operator which is inserted first onto OUTPUT.
   (c) Same operators cannot stay together in STACK so pop the operator which is pushed first onto OUTPUT.
4. If more than two operators are encountered in STACK then repeat Step-3 on the two topmost Operators present in STACK.
5. Exit.

**Numerical Example on PKR approach**

Consider the same Example: P: A + (B * C - (D / E ^ F) * G) * H
Transform P into equivalent post-fix expression in OUTPUT.

Step 1: A + (B * C - (D / E ^ F) * G) * H, read the expression from left to right.

TABLE III. CONVERSION OF INFIX TO POSTFIX EXPRESSION BY USING PKR ALGORITHM.

| Scan | STACK | Action | OUTPUT |
|---|---|---|---|
| **(1) A** | | **POP** | **A** |
| **(2) +** | **+** | **PUSH** | **A** |
| **(3) (** | **+ (** | **PUSH** | **A** |
| **(4) B** | **+ (** | **POP** | **AB** |
| **(5) *** | **+ ( *** | **PUSH** | **AB** |
| **(6) C** | **+ ( *** | **POP** | **ABC** |
| **(7) -** | **+ ( * -** | **PUSH** | **ABC** |
| **APPLY RULE** | **←yes BEMD%AS rule no→** | **NO ⇨ Pop (*)** | |
| | **+ ( -** | **POP** | **ABC*** |
| **(8) (** | **+ ( - (** | **PUSH** | **ABC*** |
| **(9) D** | **+ ( - (** | **POP** | **ABC*D** |
| **(10) /** | **+ ( - ( /** | **PUSH** | **ABC*D** |
| **(11) E** | **+ ( - ( /** | **POP** | **ABC*DE** |
| **(12) ^** | **+ ( - ( / ^** | **PUSH** | **ABC*DE** |
| **APPLY RULE** | **^ has high priority than /** | **YES ⇨ NOP** | |
| **(13) F** | **+ ( - ( / ^** | **PUSH** | **ABC*DEF** |
| **(14)** | | | |
| **(15) )** | **+ ( - *** | **POP** | **ABC*DEF^/** |
| **(16) *** | **+ ( - *** | **PUSH** | **ABC*DEF^/** |
| **(17) G** | **+ ( - ( / ^** | **POP** | **ABC*DEF^/G** |
| **APPLY RULE** | **←yes BEMD%AS rule no→** | **YES ⇨ NOP** | |
| **(18) )** | **+** | **POP** | **ABC*DEF^/G*-** |
| **(19) *** | **+ *** | **PUSH** | **ABC*DEF^/G*-** |
| **APPLY RULE** | **←yes BEMD%AS rule no→** | **YES ⇨ NOP** | |
| **(20) H** | **+ *** | **POP** | **ABC*DEF^/G*-H** |
| **(21)** | **EMPTY** | **POP** | **ABC*DEF^/G*-H*+** |

## VII. ANALYSIS BASED ON SEVERAL PARAMETERS

### (a) Performance Analysis:

TABLE IV. PERFORMANCE ANALYSIS OF EXISTING ALGORITHMS AND PKR APPROACH

| S No. | Name of the Algorithm | Number of Iterations Required | Number of the Steps needed (according to the above taken infix expression) | Complexity Discussion |
|---|---|---|---|---|
| 1. | Reverse Polish Notation (RPN) | (as per the length of string) n | 20 | Each token of infix expression will be parsed exactly once. |
| 2. | Shunting Yard Algorithm | (as per the length of string) n | 20 | Each token of infix expression will be parsed exactly once. |
| 3. | PKR Approach | (as per the length of string) n | 20 | Each token of infix expression will be parsed exactly once. |

### (b) Comparison Analysis:

TABLE V. COMPARISON ANALYSIS OF EXISTING ALGORITHMS AND PKR APPROACH.

| Name of the Algorithm | Data Structure | Space and Time Complexity | Nature (P/NP/NP-hard /NP-Complete) And Type (Recursive/Non- Recursive) | Result Display (For a particular expression) Input → Infix Expression Result → Postfix Expression |
|---|---|---|---|---|
| Reverse Polish Notation(RPN) | Stack and array or queue | Space- θ(n) Time- θ (n) | P-Time / Non-Recursive | ABC*DEF^/G*-H*+ |
| Shunting Yard Algorithm | Stack and queue | Space- θ (n) Time- θ (n) | P-Time / Non-Recursive | ABC*DEF^/G*-H*+ |
| Our proposed PKR Approach | Stack and array or queue | Space- θ (n) Time- θ (n) | P-Time / Non-Recursive | ABC*DEF^/G*-H*+ |

## VIII. PERFORMANCE

The Runtime Complexity is θ(n) and Space Complexity is also θ (n) for the Reverse Polish Notation, Shunting-yard and PKR Algorithm. We can say that our analyzed approaches give linear performance and take less memory space for conversion.

## IX. ALGORITHM OPTIMIZATION

Since there are many high level languages like C, C++, Java and .Net, can be used to optimize this procedure according to their optimization techniques.

## X. BENEFITS

These approaches are user friendly as these reduces the complexity of human effort. Because we just have to remember the rules to solve the expressions and these require very less efforts. These whole reviews will help us to add better understanding of technical concept of Infix to postfix conversion. People can take decisions to opt better algorithms for their target operations.

These are very efficient way to compute complicated calculations of expressions very easily with less number of keystrokes, by using these procedures [10]. These approaches are convenient as they have readability for humans, especially when there is a need to solve large expressions, and they are also very convenient representation for machines.

## XI. RECOMMENDATIONS

The Reverse Polish Notation, Shunting-yard and PKR conversion methods are both optimized and easier for infix to postfix conversion but also helpful for computation of expression by computer programs and these are also human friendly. So, these algorithms can be used for easily implementation and evaluation of postfix expressions and also in areas like performing calculation, parsing, logic, linguistics and lexical analysis [11]

.

## XII. LIMITATIONS

This paper discussed about existing algorithms on the basis of few expressions only. Though our discussed methods are simple but technically, some of them are limited to only postfix conversion. Also PKR algorithm does not include many logical and other operators. And PKR algorithm cannot compare logical and arithmetic operators together.

## XIII. GENERAL FUTURE SCOPE

The above algorithms can be implemented by using several Programming languages available like C, C++, Java and .Net as per their optimization policies so as to optimize. We can bring new methods in which we can invent hybrid of two data structure so as to make them simpler in implementation in computers, thus making them computer readable. And they can also be represented as a rooted tree like Abstract Syntax Tree, used in parsing in compilers.

## XIV. CONCLUSION

This paper gives an overview of existing methods for Infix to Postfix conversion. by applying some simple rules on infix expression which are highly human readable, we can make the time and space complexity linear, simpler to execute. By feeding only a few conditions to compare operators, some approaches simplify the conversion using computer program. Hence computer complexity is also reduced to an extent. These proposed algorithms are simple and can be useful in theory and practical implementation because they are human as well as computer optimized.

References:


[1] "Reverse Polish notation - Wikipedia." https://en.wikipedia.org/wiki/Reverse_Polish_notation (accessed Sep. 02, 2021).

[2] "Shunting-yard algorithm - Wikipedia." https://en.wikipedia.org/wiki/Shunting-yard_algorithm (accessed Sep. 02, 2021).

[3] "Infix notation - Wikipedia." https://en.wikipedia.org/wiki/Infix_notation (accessed Sep. 02, 2021).

[4] A. W. Burks, D. W. Warren, and J. B. Wright, "An Analysis of a Logical Machine Using Parenthesis-Free Notation," *Math. Tables Other Aids to Comput.*, vol. 8, no. 46, Apr. 1954, doi: 10.2307/2001990.

[5] "Reverse Polish Notation -- from Wolfram MathWorld." https://mathworld.wolfram.com/ReversePolishNotation.html (accessed Sep. 02, 2021).

[6] D. M. KASPRZYK, C. G. DRURY, and W. F. BIALAS, "Human behaviour and performance in calculator use with Algebraic and Reverse Polish Notation," *Ergonomics*, vol. 22, no. 9, Sep. 1979, doi: 10.1080/00140137908924675.

[7] "terminology - What is the significance of reverse polish notation? - Computer Science Stack Exchange." https://cs.stackexchange.com/questions/4666/what-is-the-significance-of-reverse-polish-notation (accessed Sep. 02, 2021).

[8] "HP calculators - Wikipedia." https://en.wikipedia.org/wiki/HP_calculators (accessed Sep. 02, 2021).

[9] "elektronika - b3-21." http://www.rskey.org/b3-21 (accessed Sep. 02, 2021).

[10] "RPN or DAL?" http://www.xnumber.com/xnumber/rpn_or_adl.htm (accessed Sep. 02, 2021).

[11] "The Future of RPN Calculators - Slashdot." https://hardware.slashdot.org/story/04/06/03/2122226/t (accessed Sep. 02, 2021).


**Appendix:**

Infix to Postfix Code:

```
/* C++ implementation to convert
infix expression to postfix*/

#include<bits/stdc++.h>
using namespace std;

//Function to return precedence of operators
int prec(char c) {
    if(c == '^')
```

```
        return 3;
    else if(c == '/' || c=='*')
        return 2;
    else if(c == '+' || c == '-')
        return 1;
    else
        return -1;
}

// The main function to convert infix expression
//to postfix expression
void infixToPostfix(string s) {

    stack<char> st; //For stack operations, we are using C++ built in stack
    string result;

    for(int i = 0; i < s.length(); i++) {
        char c = s[i];

        // If the scanned character is
        // an operand, add it to output string.
        if((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
            result += c;

        // If the scanned character is an
        // ‘(‘, push it to the stack.
        else if(c == '(')
            st.push('(');

        // If the scanned character is an ‘)’,
        // pop and to output string from the stack
        // until an ‘(‘ is encountered.
        else if(c == ')') {
            while(st.top() != '(')
            {
                result += st.top();
                st.pop();
            }
            st.pop();
        }

        //If an operator is scanned
        else {
            while(!st.empty() && prec(s[i]) <= prec(st.top())) {
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }
```

```cpp
    // Pop all the remaining elements from the stack
    while(!st.empty()) {
        result += st.top();
        st.pop();
    }

    cout << result << endl;
}

//Driver program to test above functions
int main() {
    string exp = "A+(B*C-(D/E^F)*G)*H";
    infixToPostfix(exp);
    return 0;
}
```

Assembly Code:

```
		.file "Postfux.cpp"
	.text
	.globl	stack
	.bss
	.align 32
stack:
	.space 100
	.globl	top
	.data
	.align 4
top:
	.long -1
	.text
	.globl	_Z4pushc
	.def	_Z4pushc;	.scl	2;	.type 32;	.endef
	.seh_proc	_Z4pushc
_Z4pushc:
.LFB28:
	pushq %rbp
	.seh_pushreg	%rbp
	movq	%rsp, %rbp
	.seh_setframe	%rbp, 0
	.seh_endprologue
	movl	%ecx, %eax
	movb	%al, 16(%rbp)
	movl	top(%rip), %eax
	incl	%eax
	movl	%eax, top(%rip)
	movl	top(%rip), %eax
	movslq	%eax, %rdx
	leaq	stack(%rip), %rcx
	movzbl	16(%rbp), %eax
```

```
	movb	%al, (%rdx,%rcx)
	nop
	popq	%rbp
	ret
	.seh_endproc
	.globl	_Z3popv
	.def	_Z3popv;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z3popv
_Z3popv:
.LFB29:
	pushq	%rbp
	.seh_pushreg	%rbp
	movq	%rsp, %rbp
	.seh_setframe	%rbp, 0
	.seh_endprologue
	movl	top(%rip), %eax
	cmpl	$-1, %eax
	jne	.L3
	movl	$-1, %eax
	jmp	.L4
.L3:
	movl	top(%rip), %eax
	leal	-1(%rax), %edx
	movl	%edx, top(%rip)
	cltq
	leaq	stack(%rip), %rdx
	movzbl	(%rax,%rdx), %eax
.L4:
	popq	%rbp
	ret
	.seh_endproc
	.globl	_Z8priorityc
	.def	_Z8priorityc;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z8priorityc
_Z8priorityc:
.LFB30:
	pushq	%rbp
	.seh_pushreg	%rbp
	movq	%rsp, %rbp
	.seh_setframe	%rbp, 0
	.seh_endprologue
	movl	%ecx, %eax
	movb	%al, 16(%rbp)
	cmpb	$40, 16(%rbp)
	jne	.L6
	movl	$0, %eax
	jmp	.L7
.L6:
	cmpb	$43, 16(%rbp)
	je	.L8
```

```
        cmpb  $45, 16(%rbp)
        jne   .L9
.L8:
        movl  $1, %eax
        jmp   .L7
.L9:
        cmpb  $42, 16(%rbp)
        je    .L10
        cmpb  $47, 16(%rbp)
        jne   .L11
.L10:
        movl  $2, %eax
        jmp   .L7
.L11:
        movl  $0, %eax
.L7:
        popq  %rbp
        ret
        .seh_endproc
        .def  __main;     .scl  2;    .type 32;   .endef
        .section .rdata,"dr"
.LC0:
        .ascii "Enter the expression : \0"
.LC1:
        .ascii "%s\0"
.LC2:
        .ascii "%c \0"
        .text
        .globl      main
        .def  main; .scl  2;    .type 32;   .endef
        .seh_proc   main
main:
.LFB31:
        pushq %rbp
        .seh_pushreg      %rbp
        pushq %rbx
        .seh_pushreg      %rbx
        subq  $152, %rsp
        .seh_stackalloc   152
        leaq  128(%rsp), %rbp
        .seh_setframe     %rbp, 128
        .seh_endprologue
        call  __main
        leaq  .LC0(%rip), %rcx
        call  printf
        leaq  -96(%rbp), %rax
        movq  %rax, %rdx
        leaq  .LC1(%rip), %rcx
        call  scanf
        movl  $10, %ecx
```

```
	call	putchar
	leaq	-96(%rbp), %rax
	movq	%rax, 8(%rbp)
.L20:
	movq	8(%rbp), %rax
	movzbl	(%rax), %eax
	testb	%al, %al
	je	.L13
	movq	8(%rbp), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %ecx
	movq	__imp_isalnum(%rip), %rax
	call	*%rax
	testl	%eax, %eax
	je	.L14
	movq	8(%rbp), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %edx
	leaq	.LC2(%rip), %rcx
	call	printf
	jmp	.L15
.L14:
	movq	8(%rbp), %rax
	movzbl	(%rax), %eax
	cmpb	$40, %al
	jne	.L16
	movq	8(%rbp), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %ecx
	call	_Z4pushc
	jmp	.L15
.L16:
	movq	8(%rbp), %rax
	movzbl	(%rax), %eax
	cmpb	$41, %al
	jne	.L17
.L18:
	call	_Z3popv
	movb	%al, 7(%rbp)
	cmpb	$40, 7(%rbp)
	setne	%al
	testb	%al, %al
	je	.L15
	movsbl	7(%rbp), %eax
	movl	%eax, %edx
	leaq	.LC2(%rip), %rcx
	call	printf
```

```
	jmp	.L18
.L17:
	movl	top(%rip), %eax
	cltq
	leaq	stack(%rip), %rdx
	movzbl	(%rax,%rdx), %eax
	movsbl	%al, %eax
	movl	%eax, %ecx
	call	_Z8priorityc
	movl	%eax, %ebx
	movq	8(%rbp), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %ecx
	call	_Z8priorityc
	cmpl	%eax, %ebx
	setge	%al
	testb	%al, %al
	je	.L19
	call	_Z3popv
	movsbl	%al, %eax
	movl	%eax, %edx
	leaq	.LC2(%rip), %rcx
	call	printf
	jmp	.L17
.L19:
	movq	8(%rbp), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %ecx
	call	_Z4pushc
.L15:
	incq	8(%rbp)
	jmp	.L20
.L13:
	movl	top(%rip), %eax
	cmpl	$-1, %eax
	je	.L21
	call	_Z3popv
	movsbl	%al, %eax
	movl	%eax, %edx
	leaq	.LC2(%rip), %rcx
	call	printf
	jmp	.L13
.L21:
	movl	$0, %eax
	addq	$152, %rsp
	popq	%rbx
	popq	%rbp
	ret
```

```
        .seh_endproc
        .ident      "GCC: (GNU) 9.2.0"
        .def    printf;     .scl    2;      .type   32;     .endef
        .def    scanf;      .scl    2;      .type   32;     .endef
        .def    putchar;    .scl    2;      .type   32;     .endef
```

Output:

```
cd "d:\OneDrive - std.ewubd.edu\CSE 360\" && g++ tempCodeRunnerFile.cpp -
o tempCodeRunnerFile && "d:\OneDrive - std.ewubd.edu\CSE 360\"tempCodeRunnerFile
ABC*DEF^/G*-H*+
```

Infix to Prefix Code:

```
// CPP program to convert infix to prefix
#include <bits/stdc++.h>
using namespace std;

bool isOperator(char c)
{
    return (!isalpha(c) && !isdigit(c));
}

int getPriority(char C)
{
    if (C == '-' || C == '+')
        return 1;
    else if (C == '*' || C == '/')
        return 2;
    else if (C == '^')
        return 3;
    return 0;
}

string infixToPostfix(string infix)
{
    infix = '(' + infix + ')';
    int l = infix.size();
    stack<char> char_stack;
    string output;

    for (int i = 0; i < l; i++) {

        // If the scanned character is an
        // operand, add it to output.
        if (isalpha(infix[i]) || isdigit(infix[i]))
            output += infix[i];

        // If the scanned character is an
        // '(', push it to the stack.
```

```
        else if (infix[i] == '(')
            char_stack.push('(');

        // If the scanned character is an
        // ‘)’, pop and output from the stack
        // until an ‘(‘ is encountered.
        else if (infix[i] == ')') {
            while (char_stack.top() != '(') {
                output += char_stack.top();
                char_stack.pop();
            }

            // Remove '(' from the stack
            char_stack.pop();
        }

        // Operator found
        else
        {
            if (isOperator(char_stack.top()))
            {
                if(infix[i] == '^')
                {
                    while (getPriority(infix[i]) <= getPriority(char_stack.top()))
                    {
                        output += char_stack.top();
                        char_stack.pop();
                    }

                }
                else
                {
                    while (getPriority(infix[i]) < getPriority(char_stack.top()))
                    {
                        output += char_stack.top();
                        char_stack.pop();
                    }

                }

                // Push current Operator on stack
                char_stack.push(infix[i]);
            }
        }
    }
    return output;
}

string infixToPrefix(string infix)
{
```

```
    /* Reverse String
    * Replace ( with ) and vice versa
    * Get Postfix
    * Reverse Postfix * */
    int l = infix.size();

    // Reverse infix
    reverse(infix.begin(), infix.end());

    // Replace ( with ) and vice versa
    for (int i = 0; i < l; i++) {

        if (infix[i] == '(') {
            infix[i] = ')';
            i++;
        }
        else if (infix[i] == ')') {
            infix[i] = '(';
            i++;
        }
    }

    string prefix = infixToPostfix(infix);

    // Reverse postfix
    reverse(prefix.begin(), prefix.end());

    return prefix;
}

// Driver code
int main()
{
    string s = ("A+(B*C-(D/E^F)*G)*H");
    cout << infixToPrefix(s) << endl;
    return 0;
}
```

Assembly Code:

```
                  .file  "Prefux.cpp"
      .text
      .def   __main;     .scl   2;     .type 32;    .endef
      .section .rdata,"dr"
      .align 8
.LC0:
      .ascii "\12Enter an expression in infix form: \0"
.LC1:
      .ascii "The Prefix expression is: \0"
      .text
```

```
	.globl	main
	.def	main;	.scl	2;	.type	32;	.endef
	.seh_proc	main
main:
.LFB43:
	pushq	%rbp
	.seh_pushreg	%rbp
	subq	$352, %rsp
	.seh_stackalloc	352
	leaq	128(%rsp), %rbp
	.seh_setframe	%rbp, 128
	.seh_endprologue
	call	__main
	leaq	96(%rbp), %rax
	movq	%rax, %rcx
	call	_Z9initinfixP5infix
	leaq	.LC0(%rip), %rcx
	call	printf
	leaq	32(%rbp), %rax
	movq	%rax, %rcx
	call	gets
	leaq	32(%rbp), %rdx
	leaq	96(%rbp), %rax
	movq	%rax, %rcx
	call	_Z7setexprP5infixPc
	leaq	96(%rbp), %rax
	movq	%rax, %rcx
	call	_Z7convertP5infix
	leaq	.LC1(%rip), %rcx
	call	printf
	movq	96(%rbp), %rax
	movq	104(%rbp), %rdx
	movq	%rax, -96(%rbp)
	movq	%rdx, -88(%rbp)
	movq	112(%rbp), %rax
	movq	120(%rbp), %rdx
	movq	%rax, -80(%rbp)
	movq	%rdx, -72(%rbp)
	movq	128(%rbp), %rax
	movq	136(%rbp), %rdx
	movq	%rax, -64(%rbp)
	movq	%rdx, -56(%rbp)
	movq	144(%rbp), %rax
	movq	152(%rbp), %rdx
	movq	%rax, -48(%rbp)
	movq	%rdx, -40(%rbp)
	movq	160(%rbp), %rax
	movq	168(%rbp), %rdx
	movq	%rax, -32(%rbp)
	movq	%rdx, -24(%rbp)
```

```
	movq	176(%rbp), %rax
	movq	184(%rbp), %rdx
	movq	%rax, -16(%rbp)
	movq	%rdx, -8(%rbp)
	movq	192(%rbp), %rax
	movq	200(%rbp), %rdx
	movq	%rax, 0(%rbp)
	movq	%rdx, 8(%rbp)
	movq	208(%rbp), %rax
	movq	216(%rbp), %rdx
	movq	%rax, 16(%rbp)
	movq	%rdx, 24(%rbp)
	leaq	-96(%rbp), %rax
	movq	%rax, %rcx
	call	_Z4show5infix
	call	getch
	movl	$0, %eax
	addq	$352, %rsp
	popq	%rbp
	ret
	.seh_endproc
	.globl	_Z9initinfixP5infix
	.def	_Z9initinfixP5infix;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z9initinfixP5infix
_Z9initinfixP5infix:
.LFB44:
	pushq	%rbp
	.seh_pushreg	%rbp
	movq	%rsp, %rbp
	.seh_setframe	%rbp, 0
	.seh_endprologue
	movq	%rcx, 16(%rbp)
	movq	16(%rbp), %rax
	movl	$-1, 120(%rax)
	movq	16(%rbp), %rax
	movb	$0, (%rax)
	movq	16(%rbp), %rax
	addq	$50, %rax
	movb	$0, (%rax)
	movq	16(%rbp), %rax
	movl	$0, 124(%rax)
	nop
	popq	%rbp
	ret
	.seh_endproc
	.globl	_Z7setexprP5infixPc
	.def	_Z7setexprP5infixPc;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z7setexprP5infixPc
_Z7setexprP5infixPc:
.LFB45:
```

```
	pushq	%rbp
	.seh_pushreg	%rbp
	movq	%rsp, %rbp
	.seh_setframe	%rbp, 0
	subq	$32, %rsp
	.seh_stackalloc	32
	.seh_endprologue
	movq	%rcx, 16(%rbp)
	movq	%rdx, 24(%rbp)
	movq	16(%rbp), %rax
	movq	24(%rbp), %rdx
	movq	%rdx, 104(%rax)
	movq	16(%rbp), %rax
	movq	104(%rax), %rax
	movq	%rax, %rcx
	call	strrev
	movq	16(%rbp), %rax
	movq	104(%rax), %rax
	movq	%rax, %rcx
	call	strlen
	movl	%eax, %edx
	movq	16(%rbp), %rax
	movl	%edx, 124(%rax)
	movq	16(%rbp), %rdx
	movq	16(%rbp), %rax
	movl	124(%rax), %eax
	cltq
	addq	%rdx, %rax
	movb	$0, (%rax)
	movq	16(%rbp), %rdx
	movq	16(%rbp), %rax
	movl	124(%rax), %eax
	cltq
	decq	%rax
	addq	%rax, %rdx
	movq	16(%rbp), %rax
	movq	%rdx, 112(%rax)
	nop
	addq	$32, %rsp
	popq	%rbp
	ret
	.seh_endproc
	.section .rdata,"dr"
.LC2:
	.ascii "\12Stack is full.\0"
	.text
	.globl	_Z4pushP5infixc
	.def	_Z4pushP5infixc;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z4pushP5infixc
_Z4pushP5infixc:
```

```
.LFB46:
	pushq	%rbp
	.seh_pushreg	%rbp
	movq	%rsp, %rbp
	.seh_setframe	%rbp, 0
	subq	$32, %rsp
	.seh_stackalloc	32
	.seh_endprologue
	movq	%rcx, 16(%rbp)
	movl	%edx, %eax
	movb	%al, 24(%rbp)
	movq	16(%rbp), %rax
	movl	120(%rax), %eax
	cmpl	$49, %eax
	jne	.L6
	leaq	.LC2(%rip), %rcx
	call	puts
	jmp	.L8
.L6:
	movq	16(%rbp), %rax
	movl	120(%rax), %eax
	leal	1(%rax), %edx
	movq	16(%rbp), %rax
	movl	%edx, 120(%rax)
	movq	16(%rbp), %rax
	movl	120(%rax), %eax
	movq	16(%rbp), %rcx
	movslq	%eax, %rdx
	movzbl	24(%rbp), %eax
	movb	%al, 50(%rcx,%rdx)
.L8:
	nop
	addq	$32, %rsp
	popq	%rbp
	ret
	.seh_endproc
	.section .rdata,"dr"
.LC3:
	.ascii "Stack is empty\0"
	.text
	.globl	_Z3popP5infix
	.def	_Z3popP5infix;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z3popP5infix
_Z3popP5infix:
.LFB47:
	pushq	%rbp
	.seh_pushreg	%rbp
	movq	%rsp, %rbp
	.seh_setframe	%rbp, 0
	subq	$48, %rsp
```

```
	.seh_stackalloc	48
	.seh_endprologue
	movq	%rcx, 16(%rbp)
	movq	16(%rbp), %rax
	movl	120(%rax), %eax
	cmpl	$-1, %eax
	jne	.L10
	leaq	.LC3(%rip), %rcx
	call	puts
	movl	$-1, %eax
	jmp	.L11
.L10:
	movq	16(%rbp), %rax
	movl	120(%rax), %eax
	movq	16(%rbp), %rdx
	cltq
	movzbl	50(%rdx,%rax), %eax
	movb	%al, -1(%rbp)
	movq	16(%rbp), %rax
	movl	120(%rax), %eax
	leal	-1(%rax), %edx
	movq	16(%rbp), %rax
	movl	%edx, 120(%rax)
	movzbl	-1(%rbp), %eax
.L11:
	addq	$48, %rsp
	popq	%rbp
	ret
	.seh_endproc
	.globl	_Z7convertP5infix
	.def	_Z7convertP5infix;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z7convertP5infix
_Z7convertP5infix:
.LFB48:
	pushq	%rbp
	.seh_pushreg	%rbp
	pushq	%rbx
	.seh_pushreg	%rbx
	subq	$56, %rsp
	.seh_stackalloc	56
	leaq	128(%rsp), %rbp
	.seh_setframe	%rbp, 128
	.seh_endprologue
	movq	%rcx, -48(%rbp)
.L30:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	testb	%al, %al
	je	.L13
```

```
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$32, %al
	je	.L14
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$9, %al
	jne	.L15
.L14:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	leaq	1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 104(%rax)
	jmp	.L16
.L15:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	subl	$48, %eax
	cmpl	$9, %eax
	setbe	%al
	movzbl	%al, %eax
	testl	%eax, %eax
	jne	.L20
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %ecx
	movq	__imp_isalpha(%rip), %rax
	call	*%rax
	testl	%eax, %eax
	je	.L18
.L20:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	subl	$48, %eax
	cmpl	$9, %eax
	setbe	%al
	movzbl	%al, %eax
	testl	%eax, %eax
	jne	.L19
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
```

```
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %ecx
	movq	__imp_isalpha(%rip), %rax
	call	*%rax
	testl	%eax, %eax
	je	.L18
.L19:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movq	-48(%rbp), %rdx
	movq	112(%rdx), %rdx
	movzbl	(%rax), %eax
	movb	%al, (%rdx)
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	leaq	1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 104(%rax)
	movq	-48(%rbp), %rax
	movq	112(%rax), %rax
	leaq	-1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 112(%rax)
	jmp	.L20
.L18:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$41, %al
	jne	.L21
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %edx
	movq	-48(%rbp), %rcx
	call	_Z4pushP5infixc
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	leaq	1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 104(%rax)
.L21:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$42, %al
	je	.L22
	movq	-48(%rbp), %rax
```

```
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$43, %al
	je	.L22
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$47, %al
	je	.L22
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$37, %al
	je	.L22
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$45, %al
	je	.L22
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$36, %al
	jne	.L23
.L22:
	movq	-48(%rbp), %rax
	movl	120(%rax), %eax
	cmpl	$-1, %eax
	je	.L24
	movq	-48(%rbp), %rcx
	call	_Z3popP5infix
	movb	%al, -81(%rbp)
.L26:
	movsbl	-81(%rbp), %eax
	movl	%eax, %ecx
	call	_Z8priorityc
	movl	%eax, %ebx
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %ecx
	call	_Z8priorityc
	cmpl	%eax, %ebx
	setg	%al
	testb	%al, %al
	je	.L25
	movq	-48(%rbp), %rax
	movq	112(%rax), %rdx
	movzbl	-81(%rbp), %eax
```

```
	movb	%al, (%rdx)
	movq	-48(%rbp), %rax
	movq	112(%rax), %rax
	leaq	-1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 112(%rax)
	movq	-48(%rbp), %rcx
	call	_Z3popP5infix
	movb	%al, -81(%rbp)
	jmp	.L26
.L25:
	movsbl	-81(%rbp), %eax
	movl	%eax, %edx
	movq	-48(%rbp), %rcx
	call	_Z4pushP5infixc
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %edx
	movq	-48(%rbp), %rcx
	call	_Z4pushP5infixc
	jmp	.L27
.L24:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %edx
	movq	-48(%rbp), %rcx
	call	_Z4pushP5infixc
.L27:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	leaq	1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 104(%rax)
.L23:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	movzbl	(%rax), %eax
	cmpb	$40, %al
	jne	.L30
	movq	-48(%rbp), %rcx
	call	_Z3popP5infix
	movb	%al, -81(%rbp)
.L29:
	cmpb	$41, -81(%rbp)
	je	.L28
	movq	-48(%rbp), %rax
```

```
	movq	112(%rax), %rdx
	movzbl	-81(%rbp), %eax
	movb	%al, (%rdx)
	movq	-48(%rbp), %rax
	movq	112(%rax), %rax
	leaq	-1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 112(%rax)
	movq	-48(%rbp), %rcx
	call	_Z3popP5infix
	movb	%al, -81(%rbp)
	jmp	.L29
.L28:
	movq	-48(%rbp), %rax
	movq	104(%rax), %rax
	leaq	1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 104(%rax)
.L16:
	jmp	.L30
.L13:
	movq	-48(%rbp), %rax
	movl	120(%rax), %eax
	cmpl	$-1, %eax
	je	.L31
	movq	-48(%rbp), %rcx
	call	_Z3popP5infix
	movb	%al, -81(%rbp)
	movq	-48(%rbp), %rax
	movq	112(%rax), %rdx
	movzbl	-81(%rbp), %eax
	movb	%al, (%rdx)
	movq	-48(%rbp), %rax
	movq	112(%rax), %rax
	leaq	-1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 112(%rax)
	jmp	.L13
.L31:
	movq	-48(%rbp), %rax
	movq	112(%rax), %rax
	leaq	1(%rax), %rdx
	movq	-48(%rbp), %rax
	movq	%rdx, 112(%rax)
	nop
	addq	$56, %rsp
	popq	%rbx
	popq	%rbp
	ret
	.seh_endproc
```

```
	.globl	_Z8priorityc
	.def	_Z8priorityc;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z8priorityc
_Z8priorityc:
.LFB49:
	pushq	%rbp
	.seh_pushreg	%rbp
	movq	%rsp, %rbp
	.seh_setframe	%rbp, 0
	.seh_endprologue
	movl	%ecx, %eax
	movb	%al, 16(%rbp)
	cmpb	$36, 16(%rbp)
	jne	.L33
	movl	$3, %eax
	jmp	.L34
.L33:
	cmpb	$42, 16(%rbp)
	je	.L35
	cmpb	$47, 16(%rbp)
	je	.L35
	cmpb	$37, 16(%rbp)
	jne	.L36
.L35:
	movl	$2, %eax
	jmp	.L34
.L36:
	cmpb	$43, 16(%rbp)
	je	.L37
	cmpb	$45, 16(%rbp)
	jne	.L38
.L37:
	movl	$1, %eax
	jmp	.L34
.L38:
	movl	$0, %eax
.L34:
	popq	%rbp
	ret
	.seh_endproc
	.section .rdata,"dr"
.LC4:
	.ascii " %c\0"
	.text
	.globl	_Z4show5infix
	.def	_Z4show5infix;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z4show5infix
_Z4show5infix:
.LFB50:
	pushq	%rbp
```

```
	.seh_pushreg %rbp
	pushq	%rbx
	.seh_pushreg %rbx
	subq	$40, %rsp
	.seh_stackalloc	40
	leaq	128(%rsp), %rbp
	.seh_setframe	%rbp, 128
	.seh_endprologue
	movq	%rcx, %rbx
.L41:
	movq	112(%rbx), %rax
	movzbl	(%rax), %eax
	testb	%al, %al
	je	.L42
	movq	112(%rbx), %rax
	movzbl	(%rax), %eax
	movsbl	%al, %eax
	movl	%eax, %edx
	leaq	.LC4(%rip), %rcx
	call	printf
	movq	112(%rbx), %rax
	incq	%rax
	movq	%rax, 112(%rbx)
	jmp	.L41
.L42:
	nop
	addq	$40, %rsp
	popq	%rbx
	popq	%rbp
	ret
	.seh_endproc
	.ident	"GCC: (GNU) 9.2.0"
	.def	printf;	.scl	2;	.type	32;	.endef
	.def	gets;	.scl	2;	.type	32;	.endef
	.def	getch;	.scl	2;	.type	32;	.endef
	.def	strrev;	.scl	2;	.type	32;	.endef
	.def	strlen;	.scl	2;	.type	32;	.endef
	.def	puts;	.scl	2;	.type	32;	.endef
```

Output

```
cd "d:\OneDrive - std.ewubd.edu\CSE 360\" && g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile && "d:\OneDrive - std.ewubd.edu\CSE 360\"tempCodeRunnerFile
+A*-*BC*/D^EFGH
```