# EVALUATION OF EXPRESSION IN C

## Evaluation of Infix expressions

Infix notation is commonly used in arithmetic formula or statements; the operators are written in-between their operands.

Let's assume the below

- Operands are real numbers.
- Permitted operators: +,-, *, /, ^(exponentiation)
- Blanks are permitted in expression.
- Parenthesis are permitted

**Example:**
A * ( B + C ) / D

2 * (5 + 3) / 4

Output: 4

**Approach**: Use Stacks
We will use two stacks



- Operand stack: This stack will be used to keep track of numbers.
- Operator stack: This stack will be used to keep operations (+, -, *, /, ^)

**Order of precedence of operations**–

1. ^ (Exponential)
2. / *
3. + –

**Note**: brackets ( ) are used to override these rules.
Let's define the ***Process***: (will be used for the main algorithm)

1. Pop-out two values from the operand stack, let's say it is A and B.
2. Pop-out operation from operator stack. let's say it is '+'.
3. Do A + B and push the result to the operand stack.

**Algorithm:**
Iterate through given expression, one character at a time

1. If the character is an operand, push it to the operand stack.

2. If the character is an operator,
    1. If the operator stack is empty then push it to the operator stack.
    2. Else If the operator stack is not empty,
        - If the character's precedence is greater than or equal to the precedence of the stack top of the operator stack, then push the character to the operator stack.
        - If the character's precedence is less than the precedence of the stack top of the operator stack then do Process (as explained above) until character's precedence is less or stack is not empty.
3. If the character is "(", then push it onto the operator stack.
4. If the character is ")", then do ***Process*** (as explained above) until the corresponding "(" is encountered in operator stack. Now just pop out the "(".

Once the expression iteration is completed and the operator stack is not empty, do ***Process*** until the operator stack is empty. The values left in the operand stack is our final result.

Please see the walkthrough of an example below for more understanding.

| Infix Expression: 2 * (5 *(3+6)) / 15-2 | | | | |
|---|---|---|---|---|
| **Token** | **Action** | **Operand Stack** | **Operator Stack** | **Notes** |
| 2 | Push it to operand stack | 2 | | |
| * | Push it to operator stack | 2 | * | |
| ( | Push it to operator stack | 2 | ( * | |
| 5 | Push it to operand stack | 5 2 | ( * | |
| * | Push it to operator stack | 5 2 | * ( * | |
| ( | Push it to operator stack | 5 2 | ( * ( * | |
| 3 | Push it to operand stack | 3 5 2 | ( * ( * | |
| + | Push it to operator stack | 3 5 2 | + ( * ( * | |
| 6 | Push it to operand stack | 6 3 5 2 | + ( * ( * | |
| ) | Pop 6 and 3 from operand stack | 5 2 | + ( * ( * | Do process until ( is popped from operator stack |
| | Pop + from operator stack | 5 2 | ( * ( * | |
| | Do 6+3 = 9 | 5 2 | ( * ( * | |
| | Push 9 to operand stack | 9 5 2 | ( * ( * | |
| | Pop ( from operator stack | 9 5 2 | * ( * | |
| ) | Pop 9 and 5 from operand stack | 2 | * ( * | Do process until ( is popped from operator stack |
| | Pop * from operator stack | 2 | ( * | |
| | Do 9*5 = 45 | 2 | ( * | |
| | Push 45 into operand stack | 45 2 | ( * | |
| | Pop ( from operator stack | 45 2 | * | |
| / | Push / into operator stack | 45 2 | / * | / has equal precedence to * |
| 15 | Push 15 to operand stack | 15 45 2 | / * | |
| - | Pop 15 and 45 from operand stack | 2 | / * | - has lower precedence than / , do the process |
| | Pop / from operator stack | 2 | * | |
| | Do 45/15 = 3 | 2 | * | |
| | Push 3 into operand stack | 3 2 | * | |
| | Pop 3 and 2 from operand stack | | * | - has lower precedence than * , do the process |
| | Pop * from operator stack | | | |
| | Do 3*2 = 6 | | | |
| | Push 6 into operand stack | 6 | | |
| | Push - into operator stack | 6 | - | - has equal precedence to + |
| 2 | Push 2 into operand stack | 2 6 | - | |
| | Pop 2 and 6 from the operand stack | | | Given expression is iterated, do Process till operator stack is not empty, It will give the final result |
| | Pop - from operator stack | | | |
| | Do 6-2 =4 | | | |
| | Push 4 to operand stack | **4** | | |

# Evaluation of Postfix Expressions (Polish Postfix notation)

**Postfix notation** is a notation for writing arithmetic expressions in which the operands appear before their operators. There are no precedence rules to learn, and parentheses are never needed. Because of this simplicity.
Let's assume the below

- Operands are real numbers in single digits. (Read: **Evaluation of Postfix Expressions for any Number** )
- Permitted operators: +,-, *, /, ^(exponentiation)
- Blanks are NOT permitted in expression.
- Parenthesis are permitted

**Example:**
Postfix: 54+

Output: 9

Explanation: Infix expression of above postfix is: 5+ 4 which resolves to 9

Postfix: 2536+**5/2-

Output: 16

Explanation: Infix expression of above postfix is: 2 * (5 *(3+6))/5-2 which resolves to 16

**Approach**: Use Stack
**Algorithm:**
Iterate through given expression, one character at a time

1. If the character is an operand, push it to the operand stack.
2. If the character is an operator,
   1. pop an operand from the stack, say it's s1.
   2. pop an operand from the stack, say it's s2.
   3. perform **(s2 operator s1)** and push it to stack.

3. Once the expression iteration is completed, The stack will have the final result. Pop from the stack and return the result.

Please see the walkthrough of an example below for more understanding.

| **Postfix Expression : 2536+**5/2-** | | |
|---|---|---|
| **Token** | **Action** | **Stack** |
| 2 | Push **2** to stack | [2] |
| 5 | Push **5** to stack | [2, 5] |
| 3 | Push **3** to stack | [2, 5, 3] |
| 6 | Push **6** to stack | [2, 5, 3, 6] |
| + | Pop **6** from stack | [2, 5, 3] |
| | Pop **3** from stack | [2, 5] |
| | Push **3+6 =9** to stack | [2, 5, 9] |
| * | Pop **9** from stack | [2, 5] |
| | Pop **5** from stack | [2] |
| | Push **5*9=45** to stack | [2, 45] |
| * | Pop **45** from stack | [2] |
| | Pop **2** from stack | [] |
| | Push **2*45=90** to stack | [90] |
| 5 | Push **5** to stack | [90, 5] |
| / | Pop **5** from stack | [90] |
| | Pop **90** from stack | [] |
| | Push **90/5=18** to stack | [18] |
| 2 | Push **2** to stack | [18, 2] |
| - | Pop **2** from stack | [18] |
| | Pop **18** from stack | [] |
| | Push **18-2=16** to stack | [16] |
| **Result : 16** | | |

# Evaluation of Postfix Expressions (Polish Postfix notation)

Earlier we had discussed how to evaluate postfix expressions where operands are of single-digit. In this article, we will discuss how to evaluate postfix expressions for any number ( not necessarily single digit.)
**Postfix notation** is a notation for writing arithmetic expressions in which the operands appear before their operators.
Let's assume the below

- Operands are real numbers (could be multiple digits).
- Permitted operators: +,-, *, /, ^(exponentiation)
- Blanks are used as a **separator** in expression.
- Parenthesis are permitted

**Example:**
Postfix: 500 40+

Output: 540

Explanation: Infix expression of above postfix is: 500 + 40 which resolves to 540

Postfix: 20 50 3 6 + * * 300 / 2 -

Output: 28

Explanation: Infix expression of above postfix is: 20 * (50 *(3+6))/300-2 which resolves to 28

**Approach**: Use Stack
**Algorithm:**
Iterate through given expression, one character at a time

1. If the character is a digit, initialize number = 0
   - while the next character is digit

1. do number = number*10 + currentDigit
   - push number to the stack.
2. If the character is an operator,
   - pop operand from the stack, say it's s1.
   - pop operand from the stack, say it's s2.
   - perform **(s2 operator s1)** and push it to stack.
3. Once the expression iteration is completed, The stack will have the final result. pop from the stack and return the result.

Please see the walkthrough of an example below for more understanding.

| Postfix Expression : 20 50 3 6 + * * 300 / 2 - | | |
|---|---|---|
| **Token** | **Action** | **Stack** |
| 2 | Push **20** to stack | [20] |
| 5 | Push **50** to stack | [20, 50] |
| 3 | Push **3** to stack | [20, 50, 3] |
| 6 | Push **6** to stack | [20, 50, 3, 6] |
| + | Pop **6** from stack | [20, 50, 3] |
| | Pop **3** from stack | [20, 50] |
| | Push **3+6 =9** to stack | [20, 50, 9] |
| * | Pop **9** from stack | [20, 50] |
| | Pop **50** from stack | [20] |
| | Push **50*9=450** to stack | [20, 450] |
| * | Pop **450** from stack | [20] |
| | Pop **20** from stack | [] |
| | Push **20*450=9000** to stack | [9000] |
| 300 | Push **300** to stack | [9000, 300] |
| / | Pop **300** from stack | [9000] |
| | Pop **9000** from stack | [] |
| | Push **9000/300=30** to stack | [30] |
| 2 | Push **2** to stack | [30, 2] |
| - | Pop **2** from stack | [30] |
| | Pop **30** from stack | [] |
| | Push **30-2=28** to stack | [28] |
| **Result : 28** | | |

# Evaluation of Prefix Expressions (Polish Notation)

Earlier we had discussed how to evaluate prefix expression where operands are of single-digit. Here we will discuss how to evaluate prefix expression for any number ( not necessarily single digit.)

**Prefix notation** is a notation for writing arithmetic expressions in which the operands appear after their operators. Let's assume the below

- Operands are real numbers (could be multiple digits).
- Permitted operators: +,-, *, /, ^(exponentiation)
- Blanks are used as a **separator** in expression.
- Parenthesis are permitted

**Example:**
Postfix: + 500 40

Output: 540

Explanation: Infix expression of the above prefix is: 500 + 40 which resolves to 540

Postfix: - / * 20 * 50 + 3 6 300 2

Output: 28

Explanation: Infix expression of above prefix is: 20 * (50 *(3+6))/300-2 which resolves to 28



**Approach**: Use Stack
**Algorithm:**
Reverse the given expression and Iterate through it, one character at a time

1. If the character is a digit, initialize String temp;
   - while the next character is not a digit
     - do temp = temp + currentDigit
   - convert Reverse temp into Number.
   - push Number to the stack.
2. If the character is an operator,
   - pop the operand from the stack, say it's s1.
   - pop the operand from the stack, say it's s2.
   - perform **(s1 operator s2)** and push it to stack.
3. Once the expression iteration is completed, The stack will have the final result. Pop from the stack and return the result.

Please see the walkthrough of an example below for more understanding.

| Prefix Expression : - / * 20 * 50 + 3 6 300 2 | | |
|---|---|---|
| Reversed Prefix Expression: 2 003 6 3 + 05 * 02 * / - | | |
| **Token** | **Action** | **Stack** |
| 2 | Reverse **2**, Push **2** to stack | [2] |
| 003 | Reverse **003**, Push **300** to stack | [2, 300] |
| 6 | Reverse **6**, Push **6** to stack | [2, 300, 6] |
| 3 | Reverse **3**, Push **3** to stack | [2, 300, 6, 3] |
| + | Pop **3** from stack | [2, 300, 6] |
| | Pop **6** from stack | [2, 300] |
| | Push **3+6 =9** to stack | [2, 300, 9] |
| 05 | Reverse **05**, Push **50** to stack | [2, 300, 9, 50] |
| * | Pop **50** from stack | [2, 300, 9] |
| | Pop **9** from stack | [2, 300] |
| | Push **50*9=450** to stack | [2, 300, 450] |
| 02 | Reverse **02**, Push **20** to stack | [2, 300, 450, 20] |
| * | Pop **20** from stack | [2, 300, 450] |
| | Pop **450** from stack | [2, 300] |
| | Push **20*450=9000** to stack | [2, 300, 9000] |
| / | Pop **9000** from stack | [2, 300] |
| | Pop **300** from stack | [2] |
| | Push **9000/300=30** to stack | [2, 30] |
| - | Pop **30** from stack | [2] |
| | Pop **2** from stack | [] |
| | Push **30-2=28** to stack | [28] |
| **Result : 28** | | |