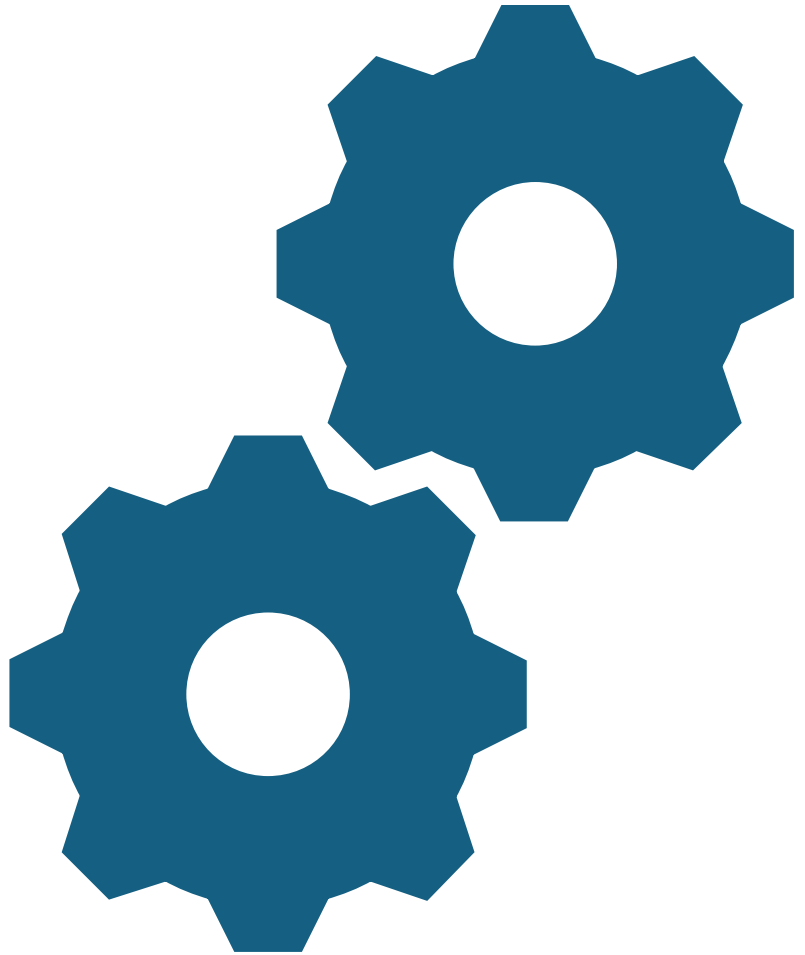
BITE1523:
COMPUTER GAME
PROGRAMMING

# Chapter 3: Stack



# Recap: Types of data structure: STACK

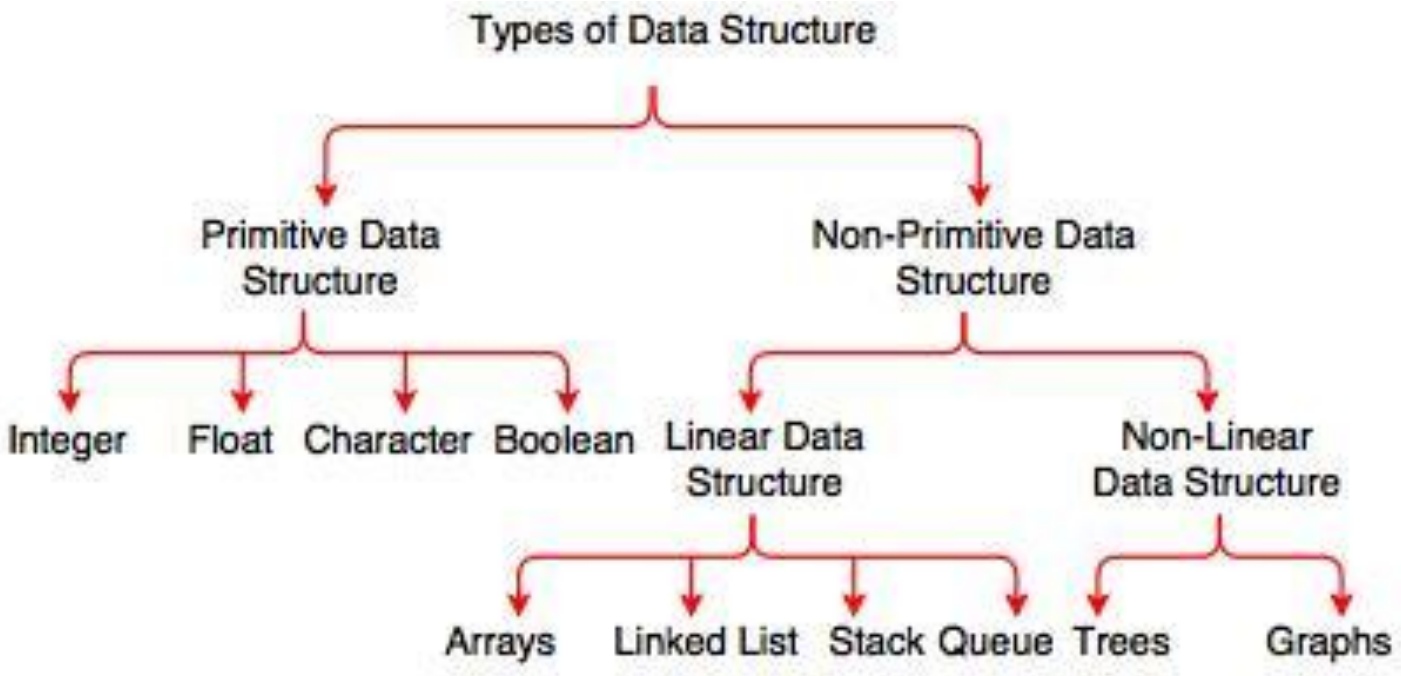



Fig. Types of Data Structure

# Outline

Stack Concept and Structure

Stack Operations

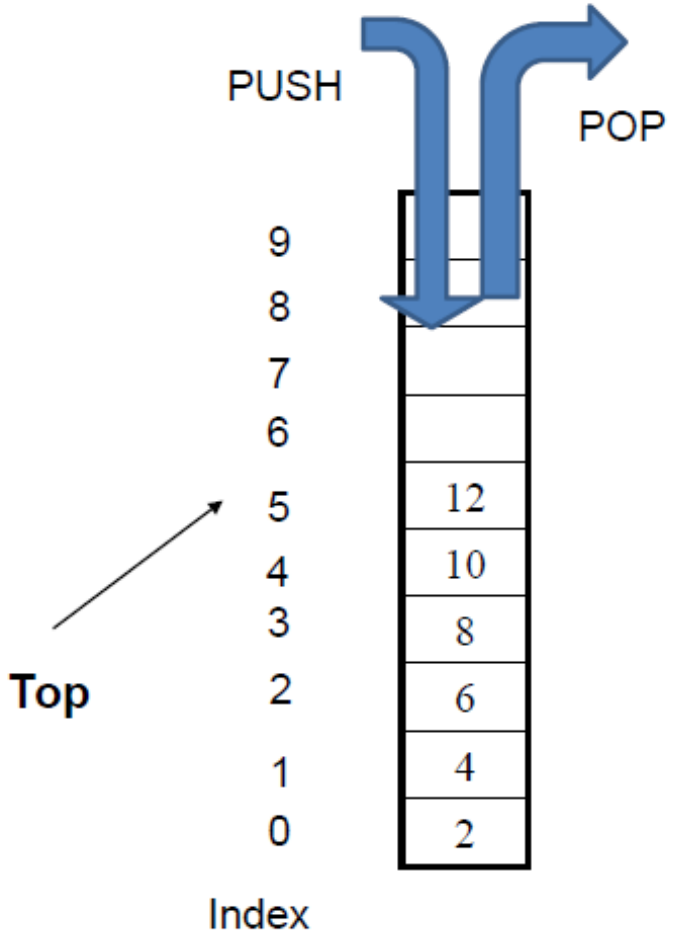
Stack Implementation using array  and linked list

# What is Stack

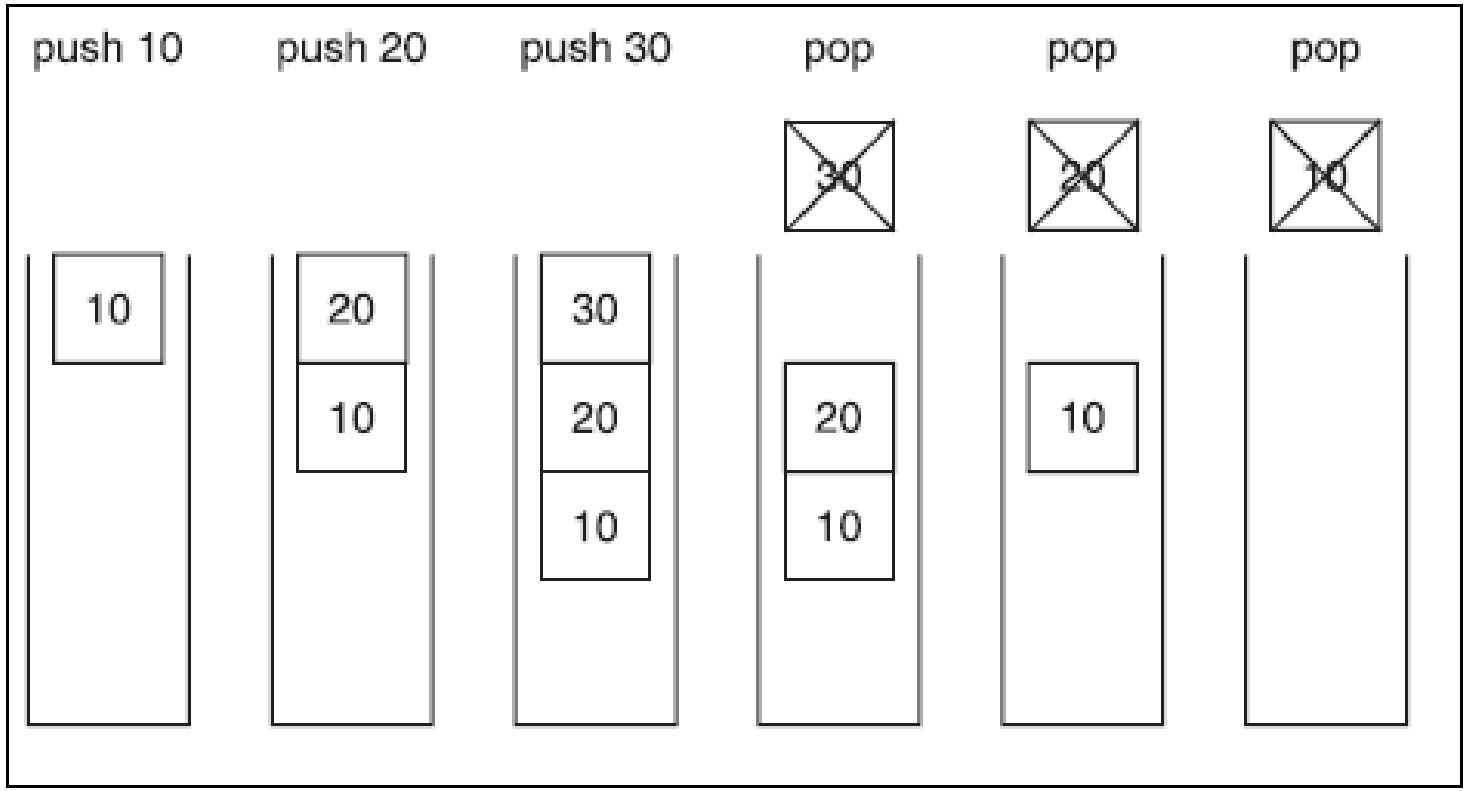
- Stack is a collection of items which is organized in a sequential manner.
- Example: stack of books or stac of plates.
- All additions and deletions are restricted at one end, called top.
- LAST IN FIRST OUT (LIFO) data structure.

# What is Stack

- Stack is an abstract data type
- Adding an entry on the top (push)
- Deleting an entry from the top (pop)
- A stack is open at one end (the top) only. You can push entry onto the top, or pop the top entry out of the stack.



Last-in First-Out (LIFO)



# Stack ADT

A stack is an abstract data type (ADT) where elements can be inserted (pushed) and removed (popped) at one end only, according to the following policies :

- **Push** - inserts a new item into the top of the stack.
- **Pop** - removes the item at the top of the stack (except when the stack is empty).
- The above insert/remove policy is also called **LIFO (Last In-First Out).**

# Stack Operations

| |
|---|
| Push(x)- places an item on the top of the stack. |
| Pop()- removes the item at the top of the stack. |
| Top()- accesses the item at the top of the stack. |
| IsEmpty()– check whether the stack is empty |
| isFull()- check whether the stack is full |
| All operations can be performed in constant time. |
| Time complexity is O(1). |

Implementation of Stack

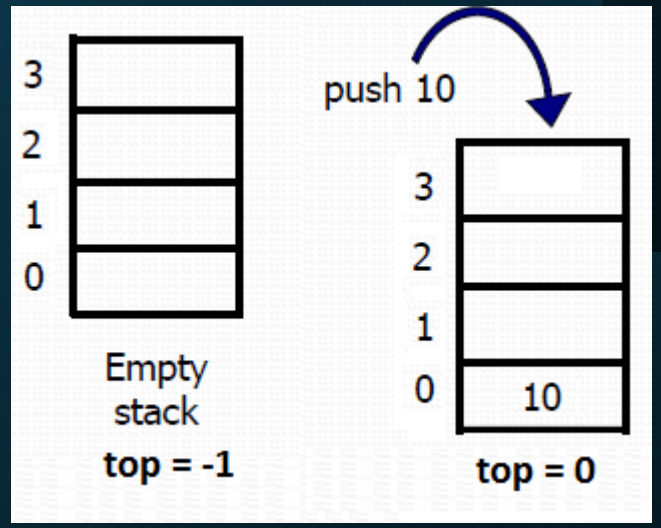
Array

- **Size of stack is fixed during declaration**
- Item can be pushed if there is some space available
- Need a variable called, top to keep track the top of a stack.
- Stack is empty when the value of Top is –1.

**Linked List**

- **Size of stack is flexible**. Item can be pushed and popped dynamically.
- Need a pointer, called top to point to top of stack.

# Implementation of Stack - Array

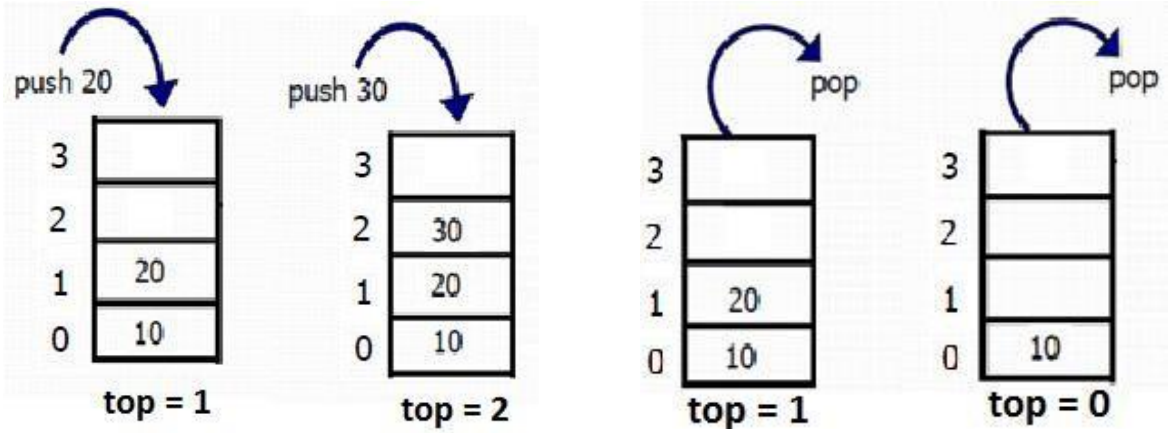



For stack in an array, the number of elements in a stack is fixed.

The initial position is -1, which means an empty stack. See diagram below.

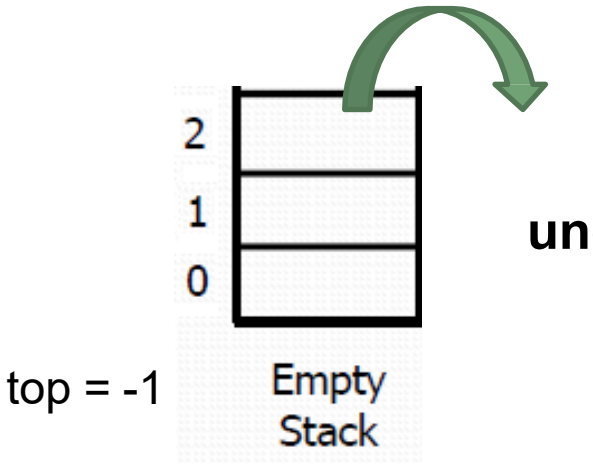
# Implementation of Stack - Array

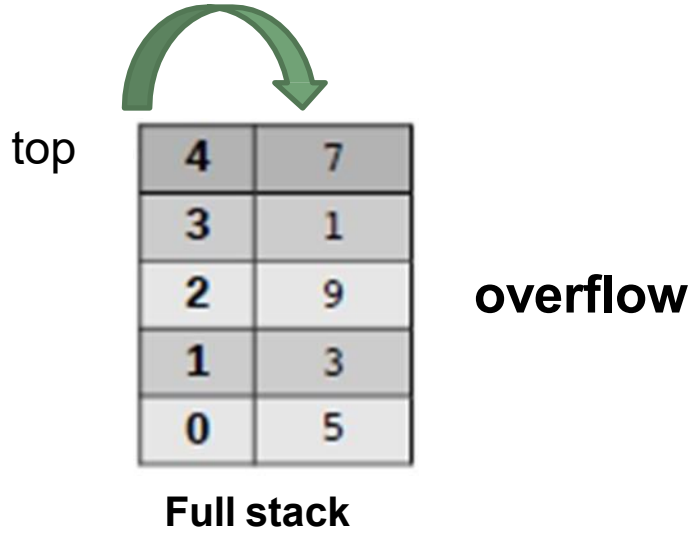
When push, top index is increased and when pop, top index is decreased.



# Implementation of Stack - Array

- Stack **underflow** happens when we try to pop an item from an empty stack, top = -1.
- Stack **overflow** happens when we try to push an item into a full stack, top = MAX-1



# Stack Array Implementation, cont...

3 things to be considered for stack with array

1. **Stack Empty :** when top is -1.
2. **Push operations:** To insert item into stack

   2 statements must be used

   ```
   top = top + 1;
   stack[top] = newitem;
   ```

3. **pop operations:** To delete item from stack.

   2 statements should be used

   Item = stack[top]; or stackTop();

   top = top – 1;

- `Item = stack[top];` statement is needed if we want to check the value to be popped.

```
#include <cstdlib>
#include <iostream>
#define MAXSIZE 31

using namespace std;

int A[MAXSIZE];
int top = -1;  void Push(int x)
{
     if (top == MAXSIZE - 1)
     {
        cout << "Error: Overflow " << endl;
        return;
     }
     A[++top] = x;
}
void Pop()
{
     if (top == -1)
     {
     cout << "Error:No element to pop "<< endl;
     return;
     }
top--;
}
```

```
int Top()
{
return A[top];
}
void Print()
{
int i;  cout << "Stack : ";
for (i = 0; i <= top; i++)
cout << " " << A[i];
cout << endl;
}
int main()
{
Push(2); Print();
Push(5); Print();
Push(10); Print();
Pop(); Print();
Push(12); Print();
}
```

# Stack using Linked List

# Stack Linked List Implementation

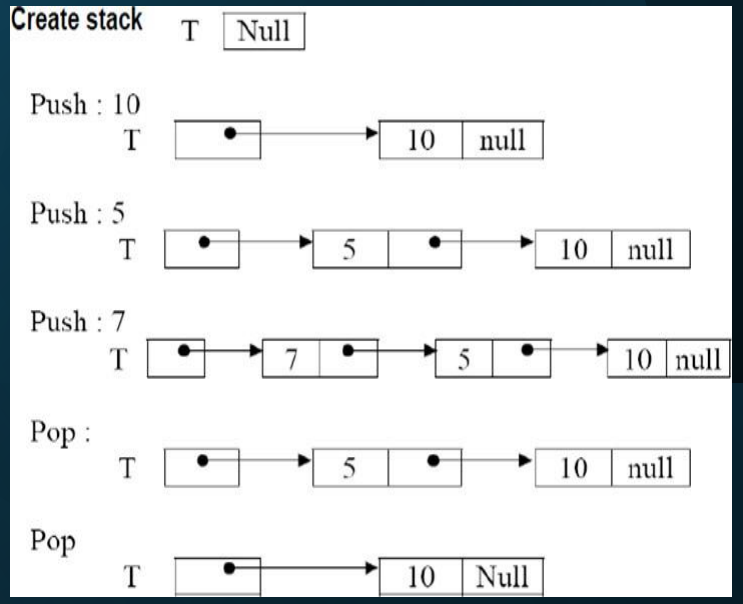
Stack Linked List (Using Pointers)

Stack in an array is much easier and efficient because number of elements is fixed.

The problem is when the stack size is unknown and this could caused the stack to be *overflow*.

So, stack linked list (pointers) is suitable to use as long as the computer memory is not full.

# Implementation Using Linked List

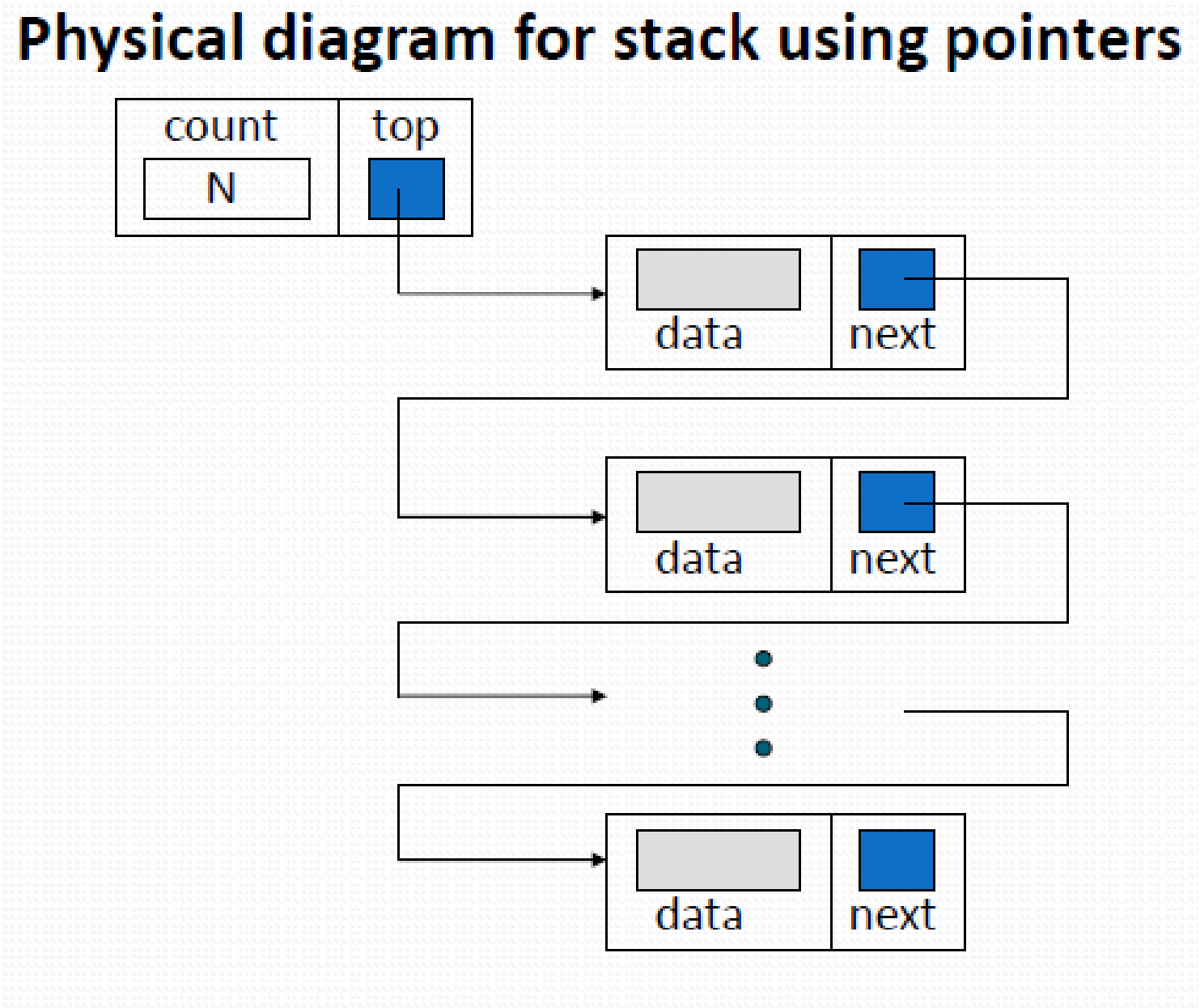



**Size of stack is flexible.** Item can be pushed and popped dynamically.

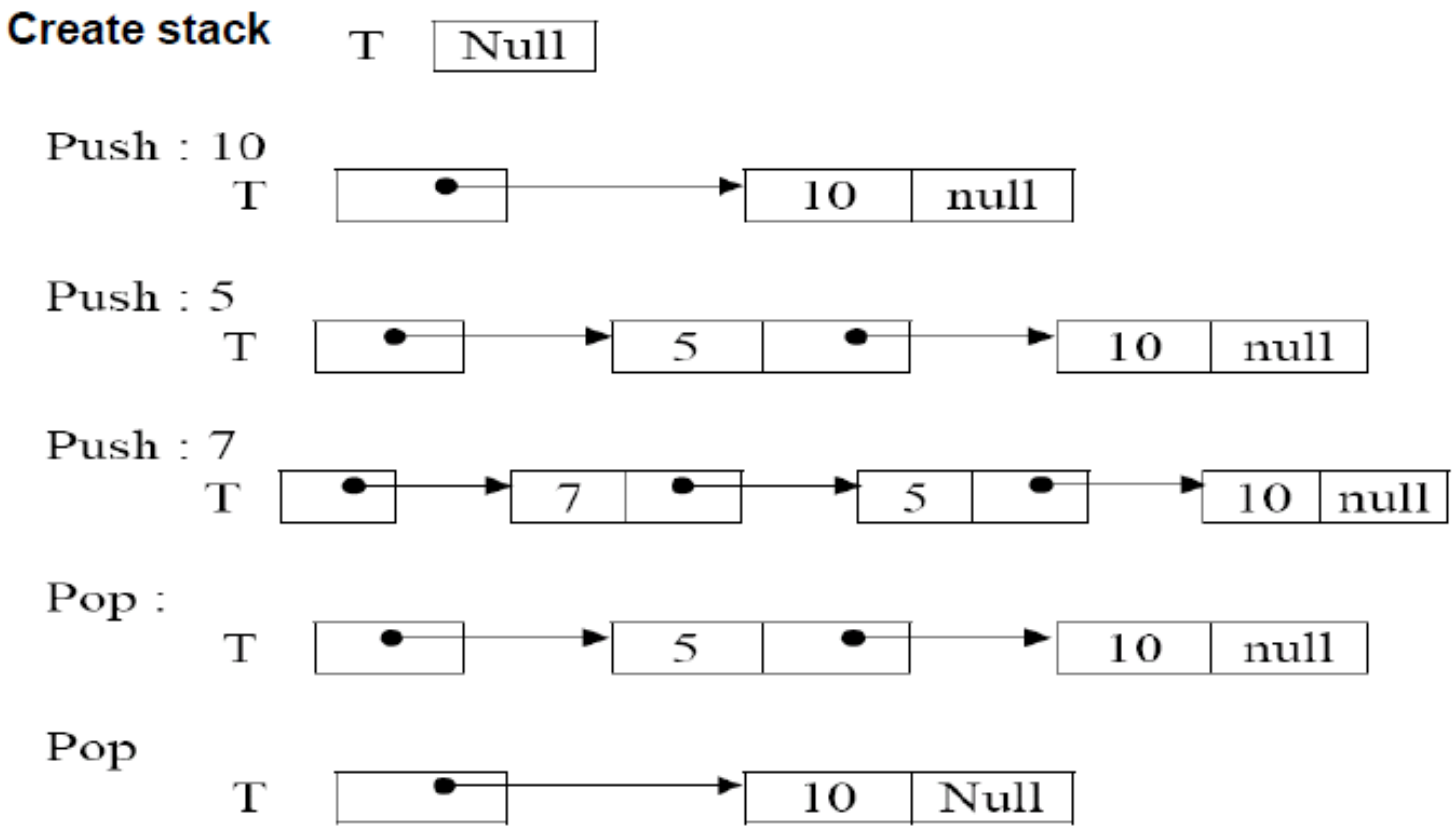
Need a pointer, called top to point to top of stack.

Insert / Delete is at beginning. Time Complexity – O(1)

# Stack Linked List, cont...



# Linked List, cont...



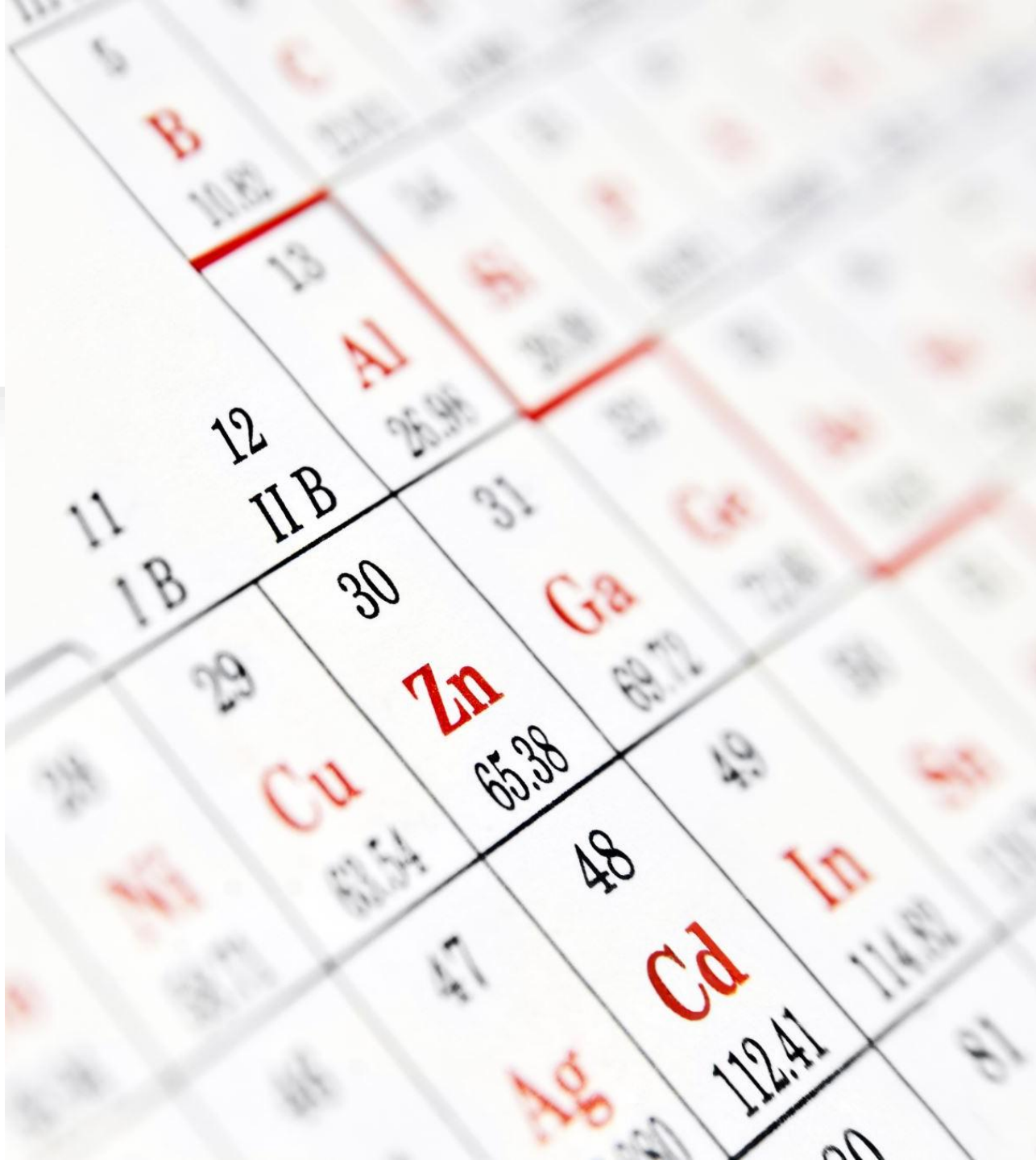
# Basic Implementation in C++ (LL)

```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

struct Node {
    int data;
    Node* next;
};

Node* top = NULL;

void Push(int x) {
    Node* temp = new Node();
    temp->data = x;
    temp->next = top;
    top = temp;
}

void Pop() {
    if (top == NULL) return;
    Node* temp = top;
    top = top->next;
    free(temp);
}
```

```cpp
void Print() {
    Node* temp = top;
    cout << "List is: ";
    while (temp != NULL) {
        cout << temp->data << "\t";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Push(2);
    Print();
    Push(5);
    Print();
    Push(10);
    Print();
    Pop();
    Print();
    Push(12);
    Print();
    return 0;
}
```

# Application Using Stack

- Reverse a String (Reverse word) – e.g to find palindrome words ( example : radar, hannah, racecar, katak)
- System software and Compiler design
  - Undo Command – go back to previous command
- Parenthesis matching.
    - ((a+b)×(c+d)×e)–(f+g)



# Calculate an arithmetic expression using RPN

Reverse Polish Notation (RPN) is a postfix, parenthesis-free, mathematical notation where the operator follows their operands. (2 steps)

1. **Conversion infix to postfix expression**
   Arithmetic expression => (a + b) × (c + d) **(infix)**
   RPN expression => ab+cd+× **(postfix)**
2. **Calculate Postfix**

# WHAT IS INFIX AND POSTFIX? Contoh 3 + 4

**INFIX**

- Operator (+) duduk di tengah antara dua nombor.
- Senang baca, tapi susah sikit untuk komputer evaluate sebab kena fikir pasal keutamaan operasi (precedence) dan bracket (parentheses).

**POSTFIX**

- Kenapa guna Postfix?

👉 **Tak perlu bracket langsung**
👉 **Komputer suka** sebab senang evaluate guna **Stack**

**Contoh: 34+** membawa maksud 3+4

# Step 1 : Infix to Postfix expression

- General Idea

To convert infix to postfix, we usually use the Shunting Yard Algorithm (by Dijkstra). It uses a stack to temporarily hold operators.

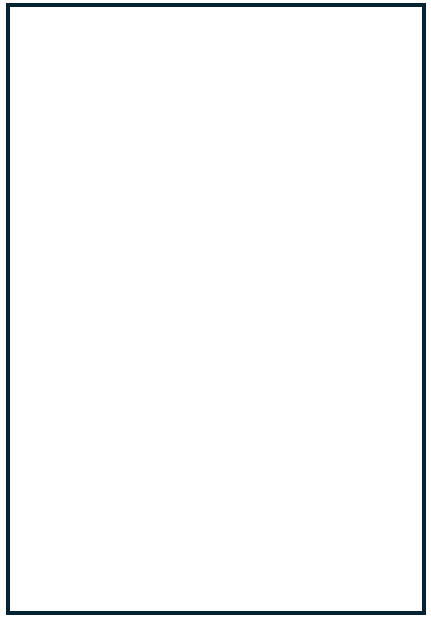
- Operator Precedence (Very Important)

When converting, we need to respect:

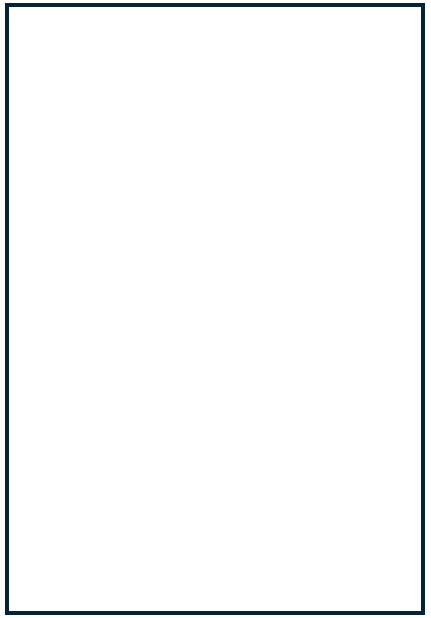
- **()** → Highest precedence
- ***** and **/** → higher than **+** and **–**
- Operators of same precedence are usually left-to-right (left-associative)

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**



**Output**

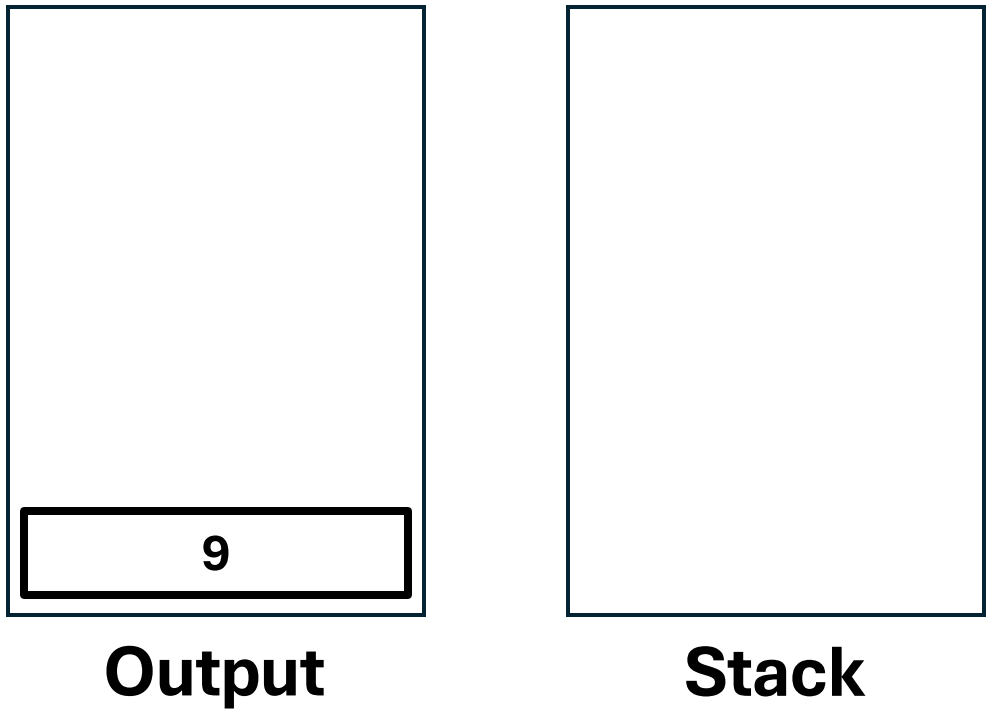


**Stack**

**Token = NULL**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Token: 9**

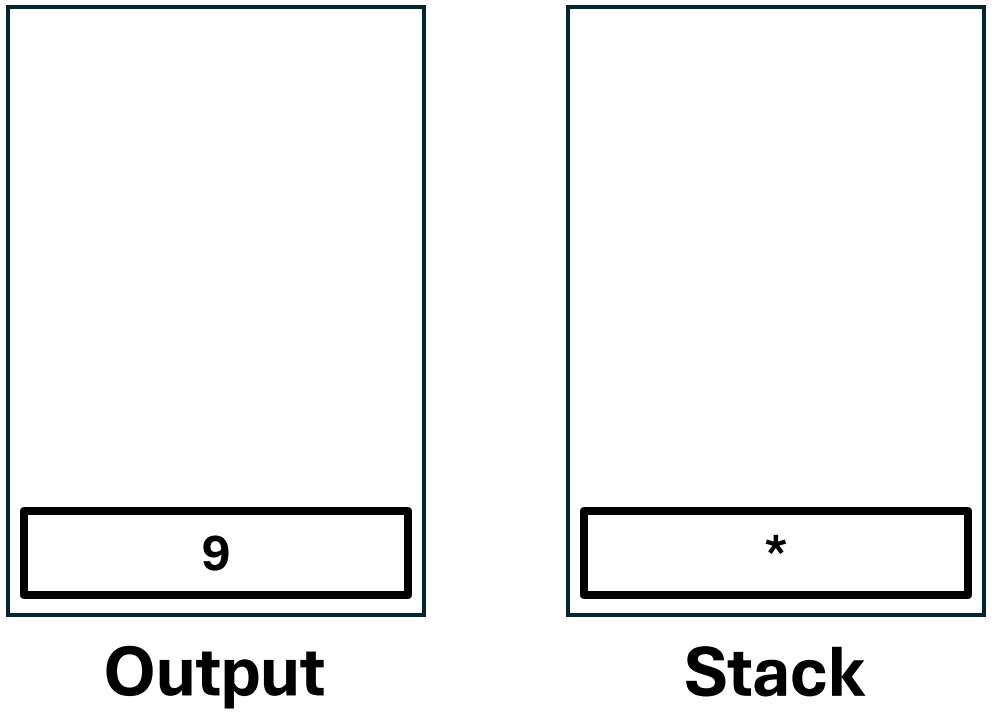
**→ it's a number**

**→ add to output**

**Output = [9]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Token: ***

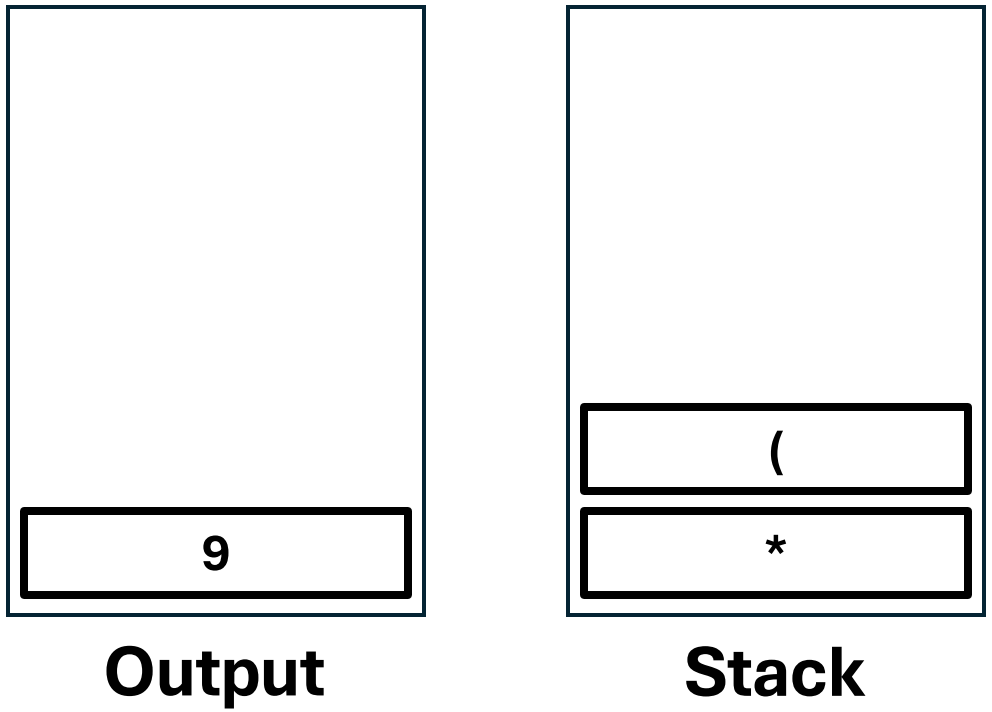
**→ operator**

**→ push to stack**

**Stack = [*]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Token: (**

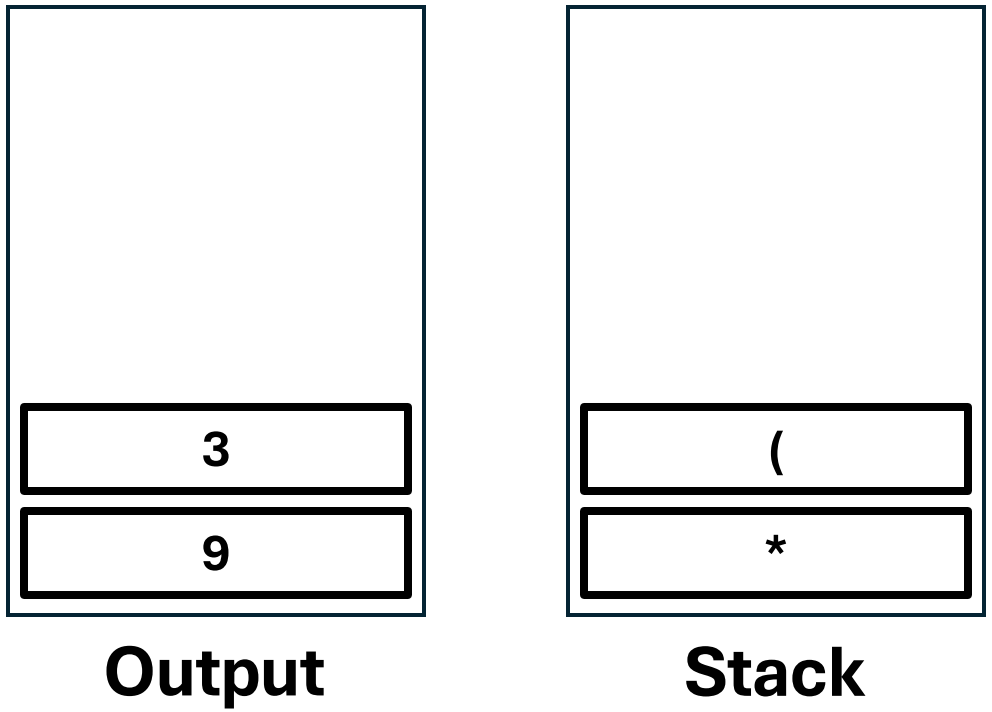
**→ operator**

**→ push to stack**

**Stack = [*, (]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Token: 3**

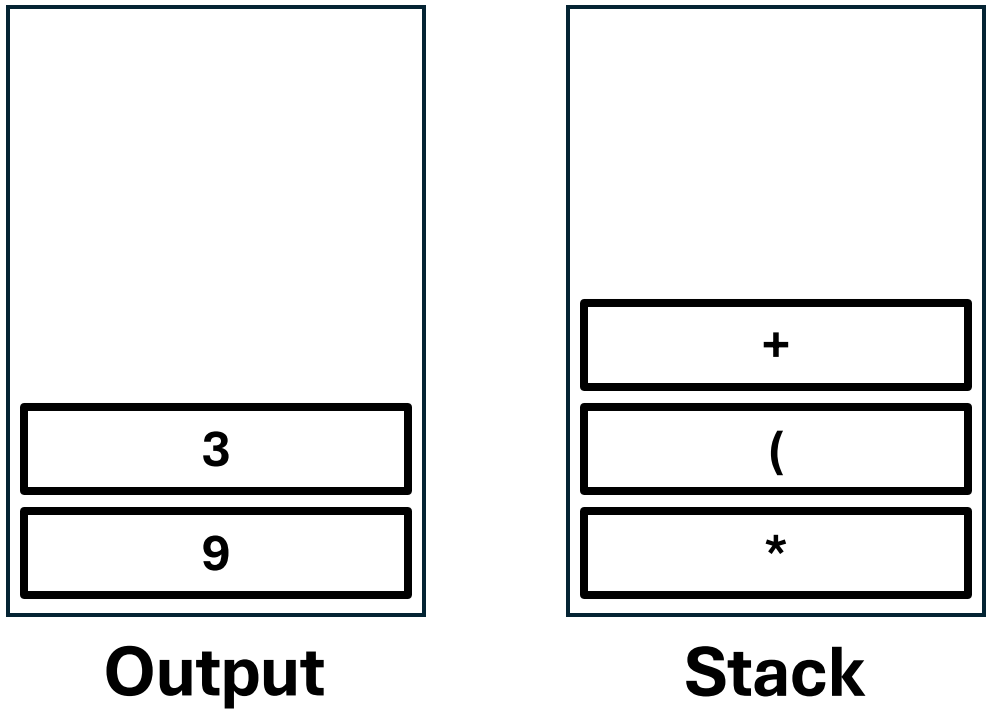
**→ it's a number**

**→ add to output**

**Output = [9,3]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Token: +**

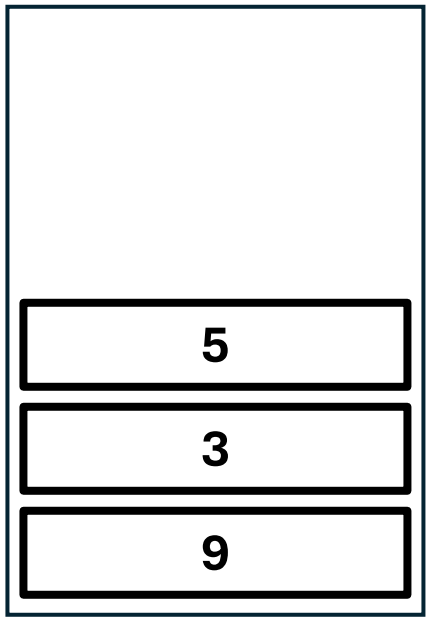
**→ operator**

**→ push to stack**

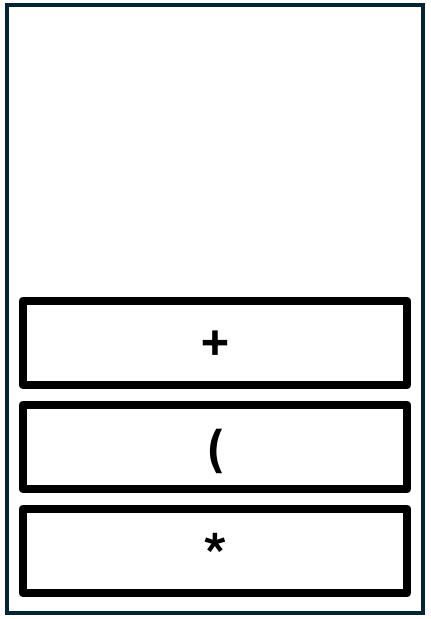
**Stack = [*, (,+]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Output**




**Stack**

**Token: 5**

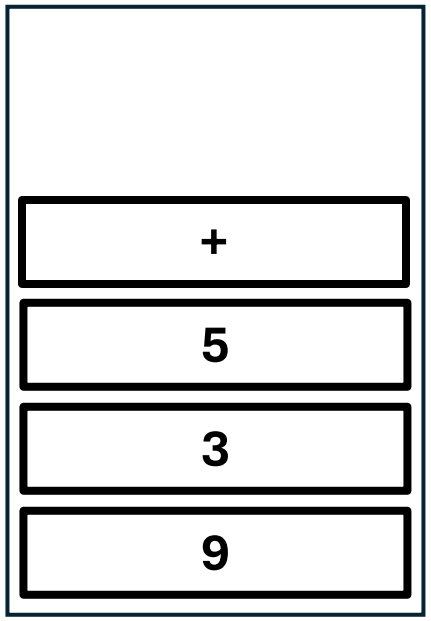
**→ it's a number**

**→ add to output**

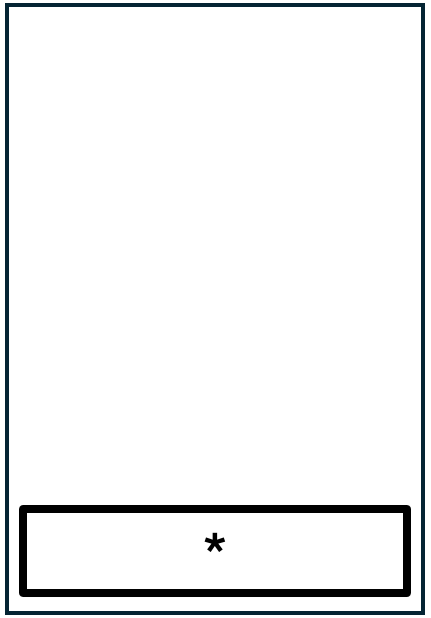
**Output = [9,3,5]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Output**




**Stack**

**Token: )**

**→ pop until (**

**→ Pop +**

**→ add to output**

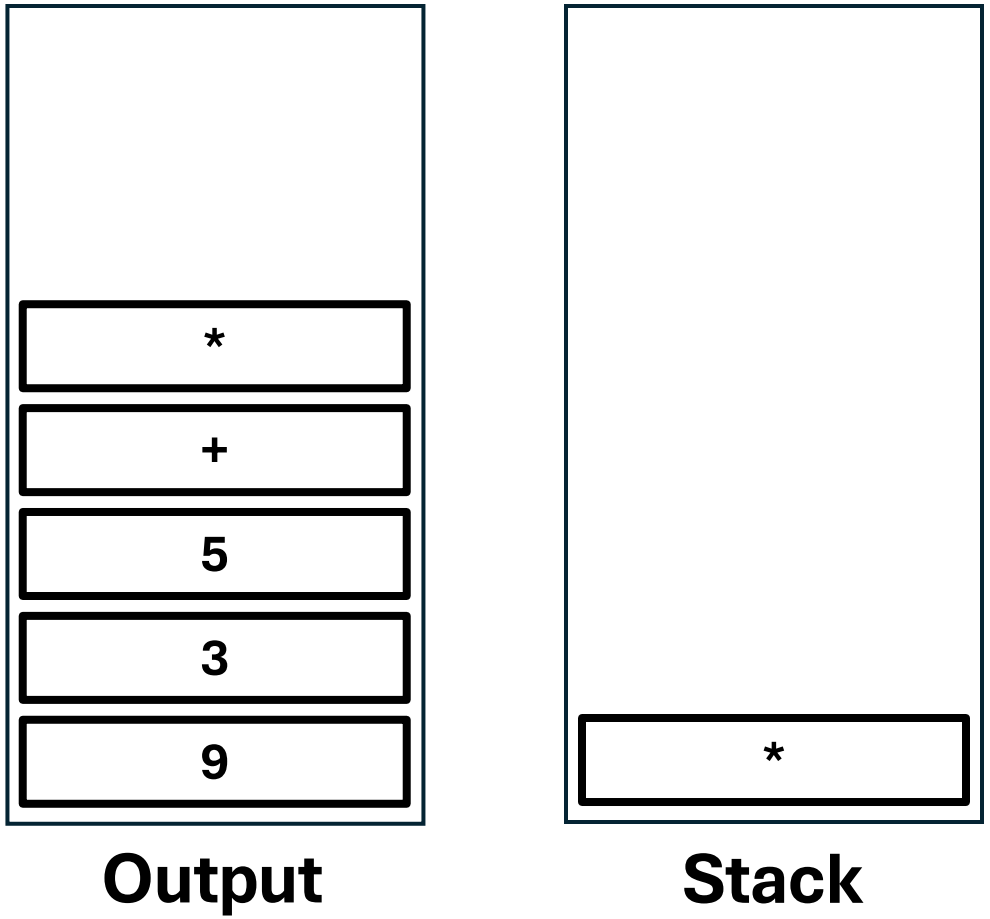
**→ Pop ( → discard**

**Output = [9, 3, 5, +]**

**Stack = [*]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Token: ***

**→ operator**

**→ precedence of * is same as top of stack ***

**→ pop top * from stack**

**→ add to output**

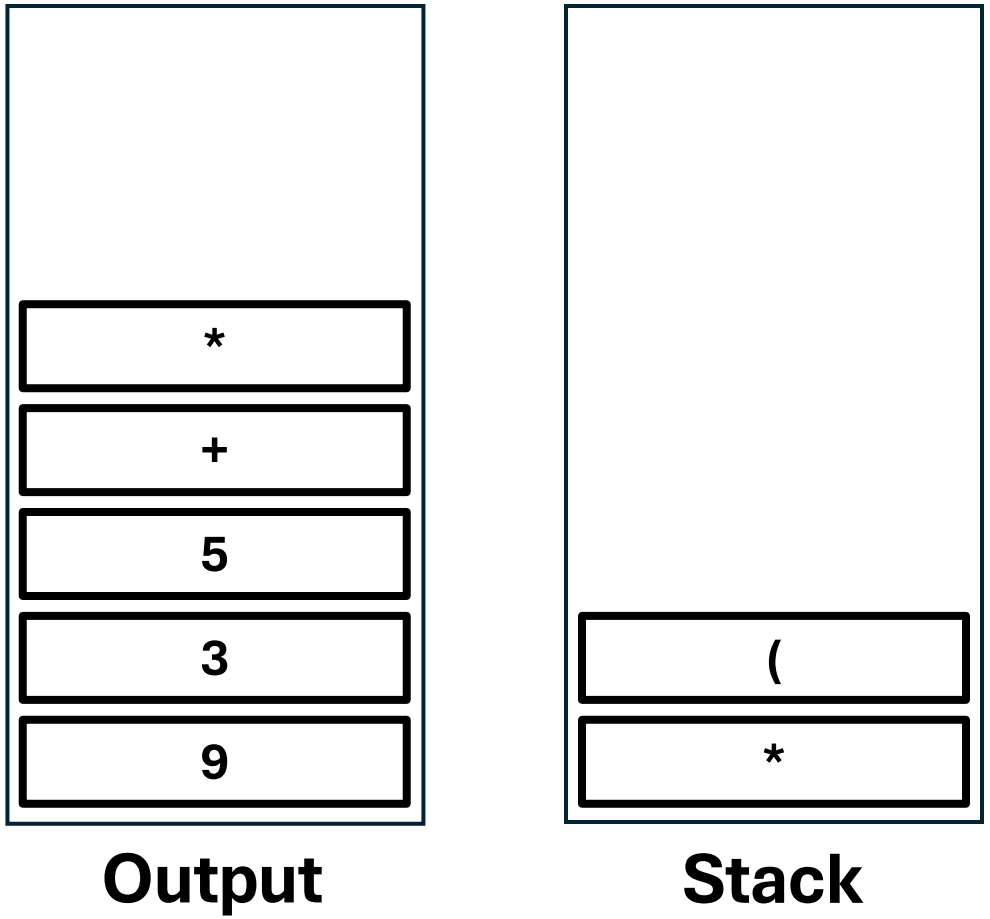
**→ push current * to stack**

**Output = [9, 3, 5, +, *]**

**Stack = [*]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Token: (**

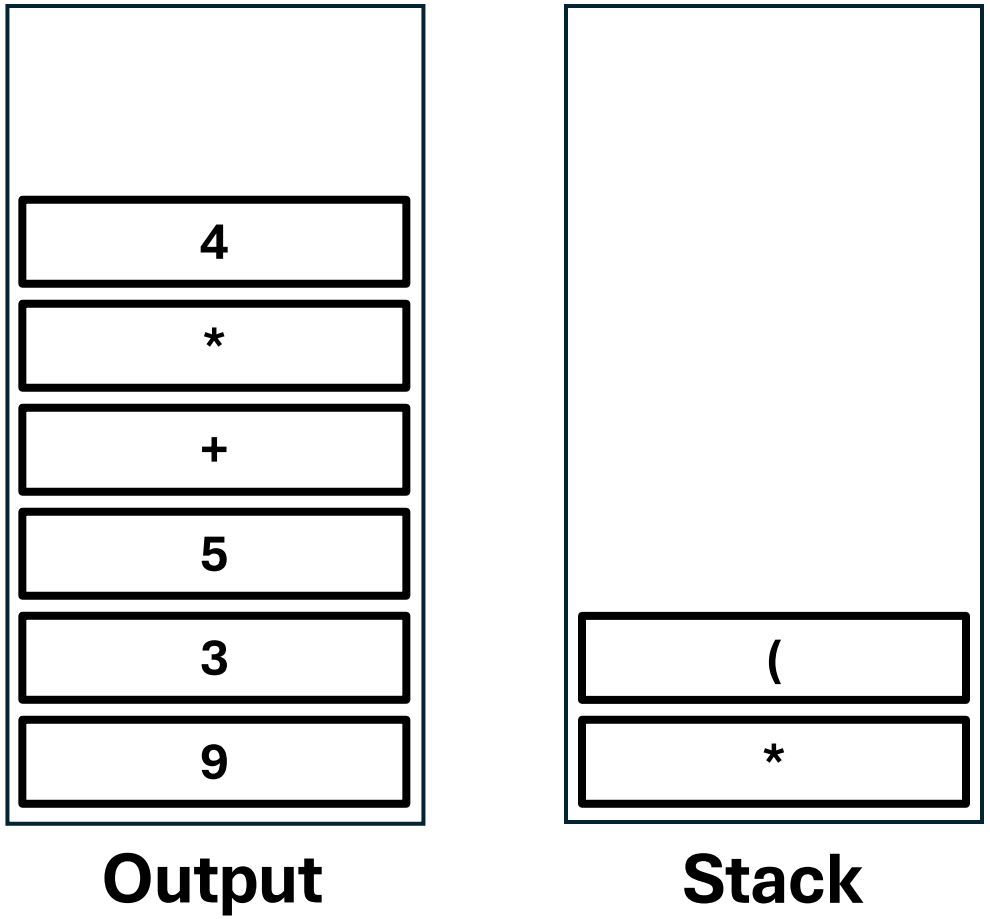
**→ operator**

**→ push to stack**

**Stack = [*, (]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Token: 4**

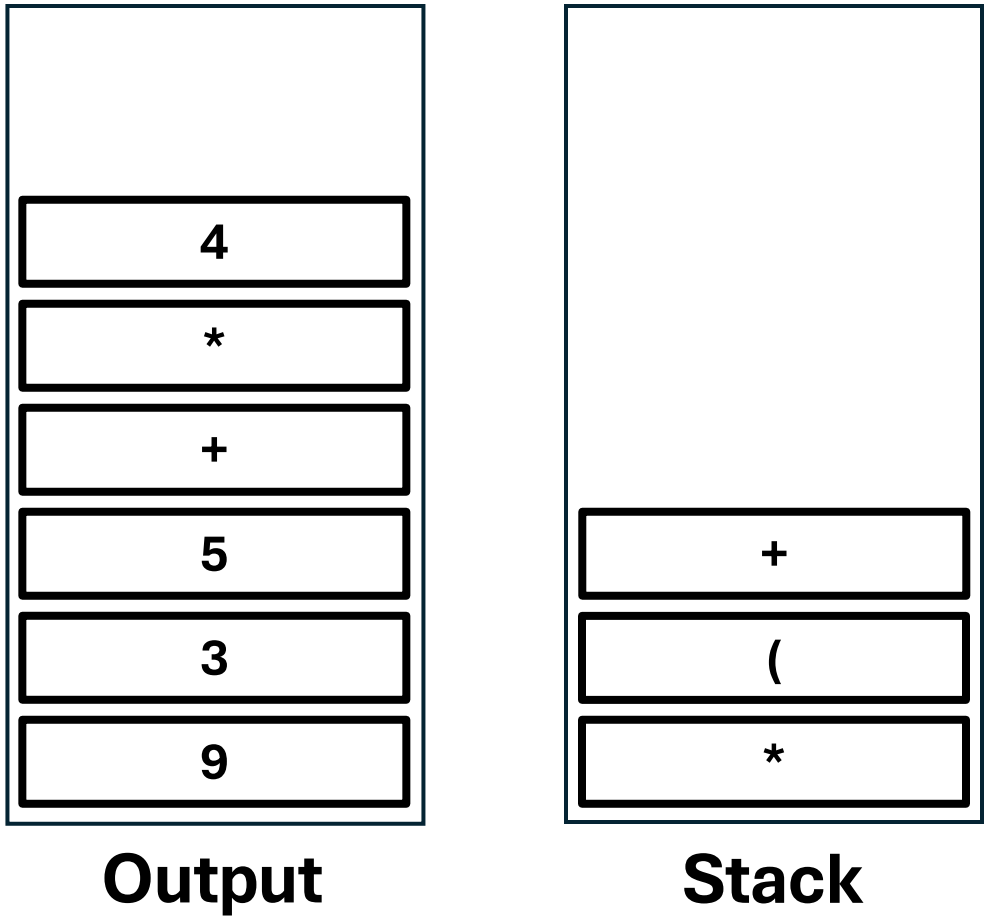
**→ it's a number**

**→ add to output**

**Output = [9,3,5,+,*,4]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**




**Token: +**

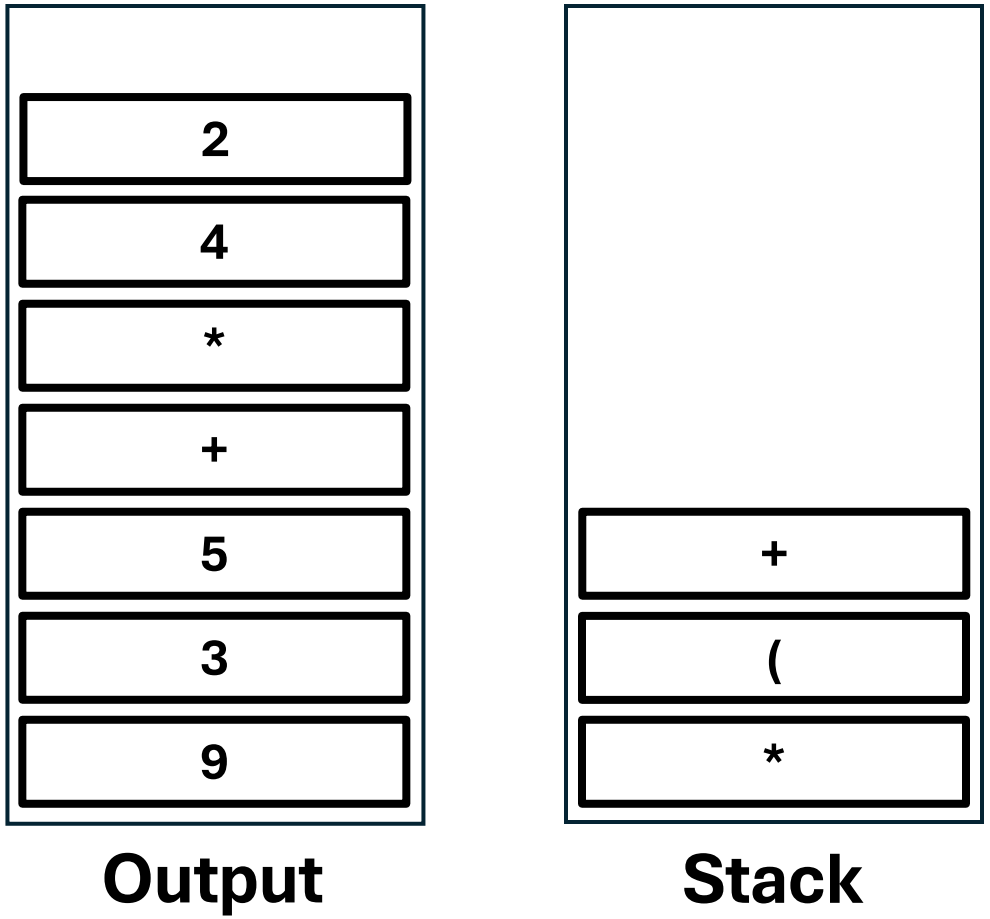
**→ operator**

**→ push to stack**

**Stack = [*, (,+]**

# Infix to Postfix expression

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**



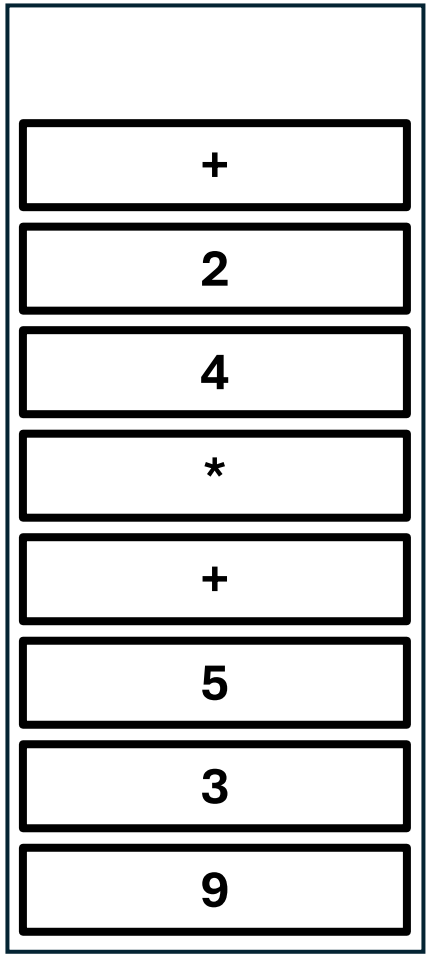

**Token: 2**

**→ it's a number**

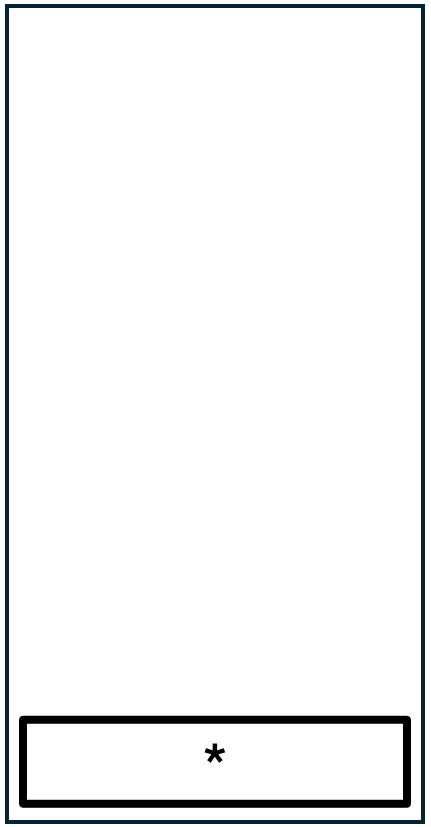
**→ add to output**

**Output = [9,3,5,+,*,4,2]**

# Infix to Postfix expression




Output




Stack

**Start** **9 * ( 3 + 5 ) * (4 + 2) = ?**

**Token: )**

**→ pop until (**

**→ Pop +**

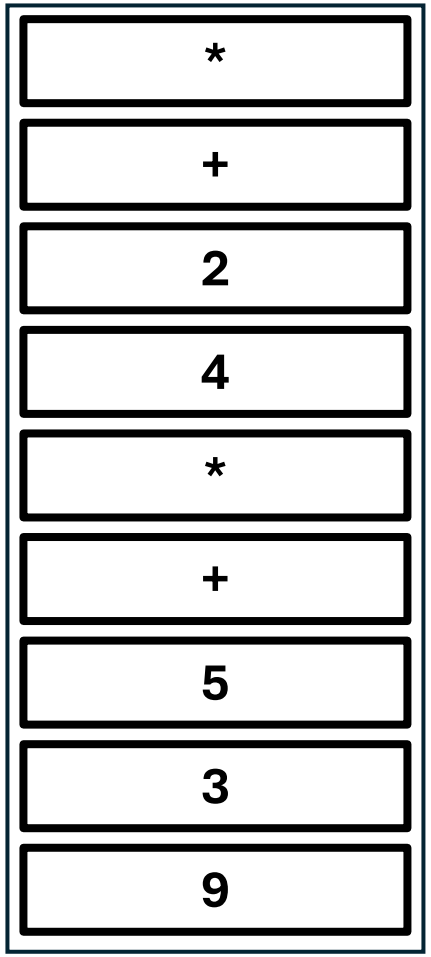
**→ add to output**

**→ Pop ( → discard**

**Output = [9, 3, 5, +, *, 4, 2, +]**

**Stack = [*]**

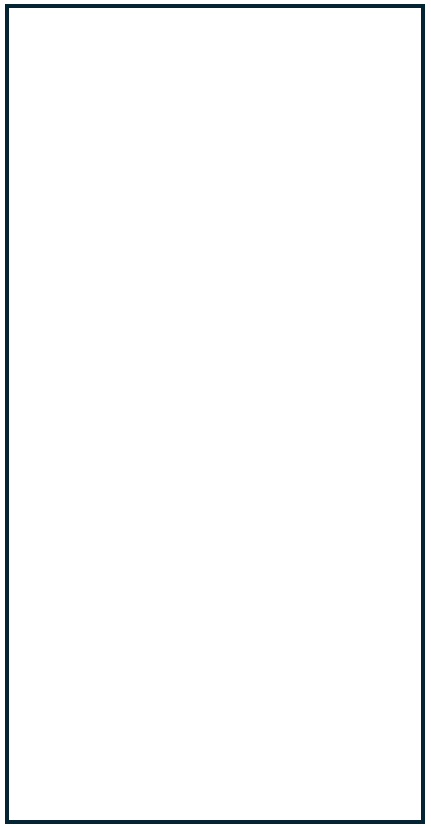
# Infix to Postfix expression




Output

**Start**

**9 * ( 3 + 5 ) * (4 + 2) = ?**



Stack

**Pop remaining operators from the stack**

**→ pop ***

**→ add to output**

**Output = [9, 3, 5, +, *, 4, 2, +, *]**

# 🔷 Step-by-Step Evaluation Using Stack

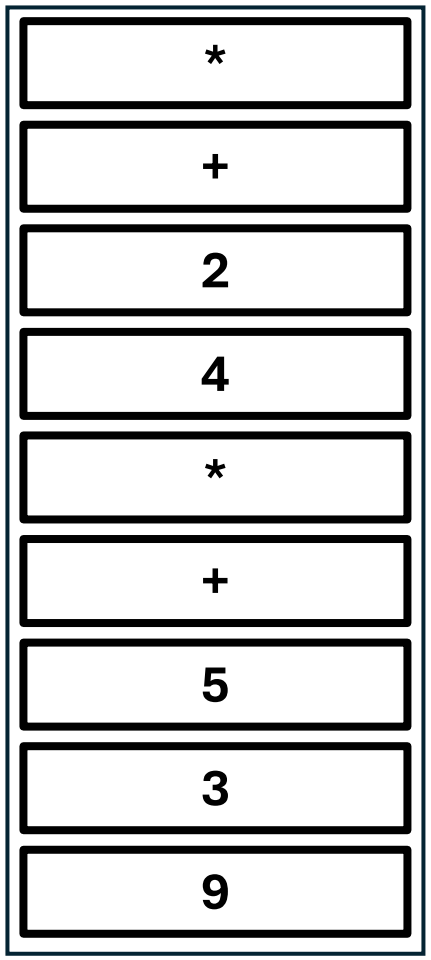
## [9, 3, 5, +, *, 4, 2, +, *]

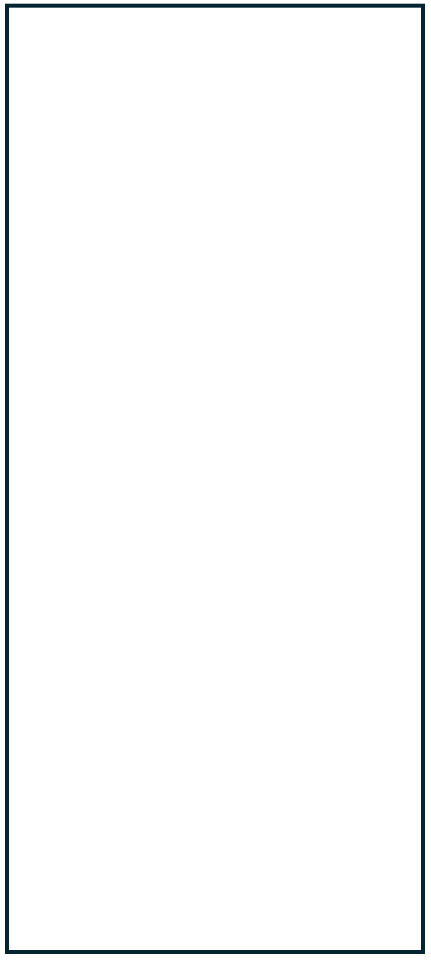
We’ll use a stack to store intermediate numbers.

**Start with empty stack.**

# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**



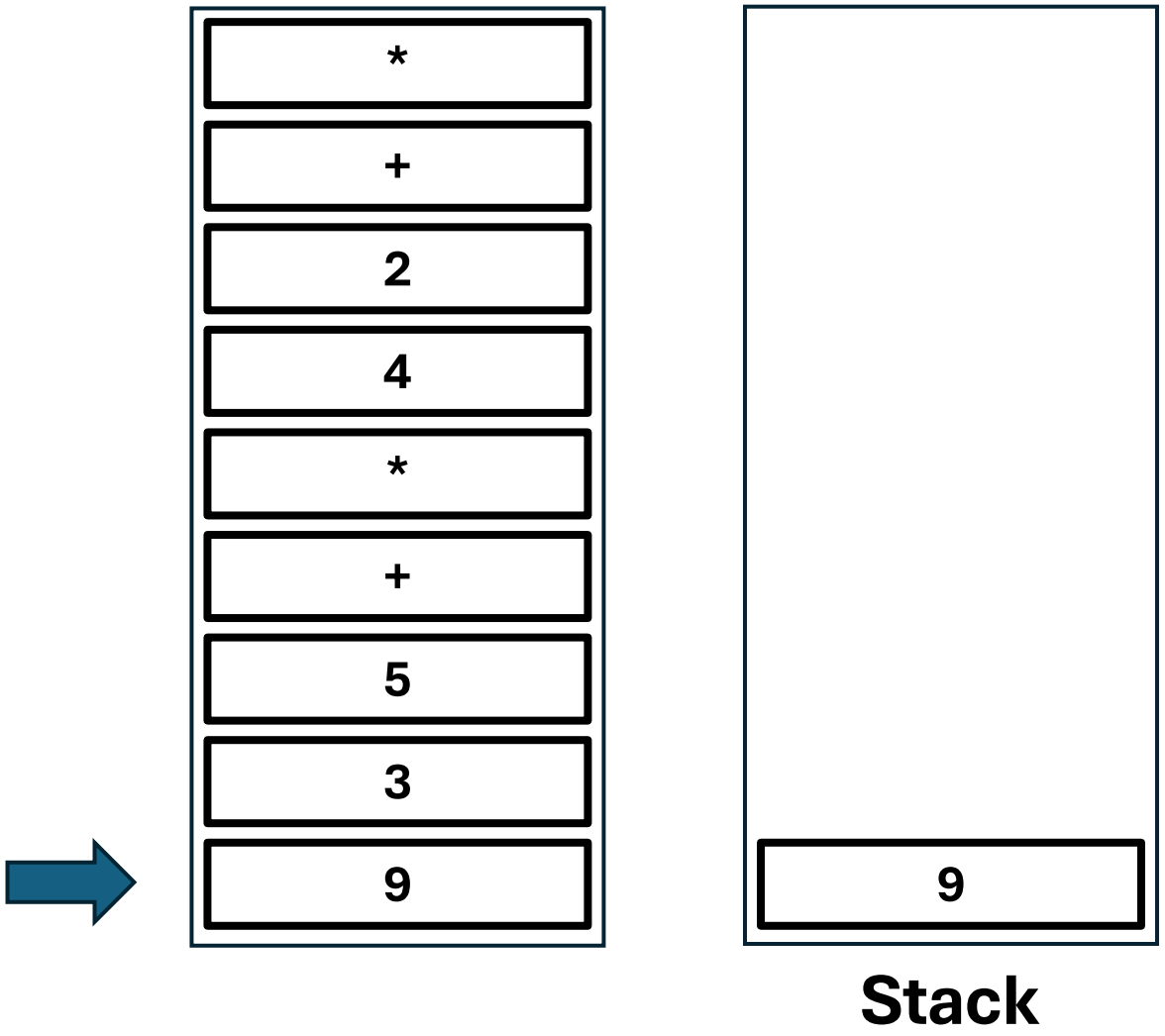



**Stack**

**Start with Empty Stack**

# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**




**Token: 9**

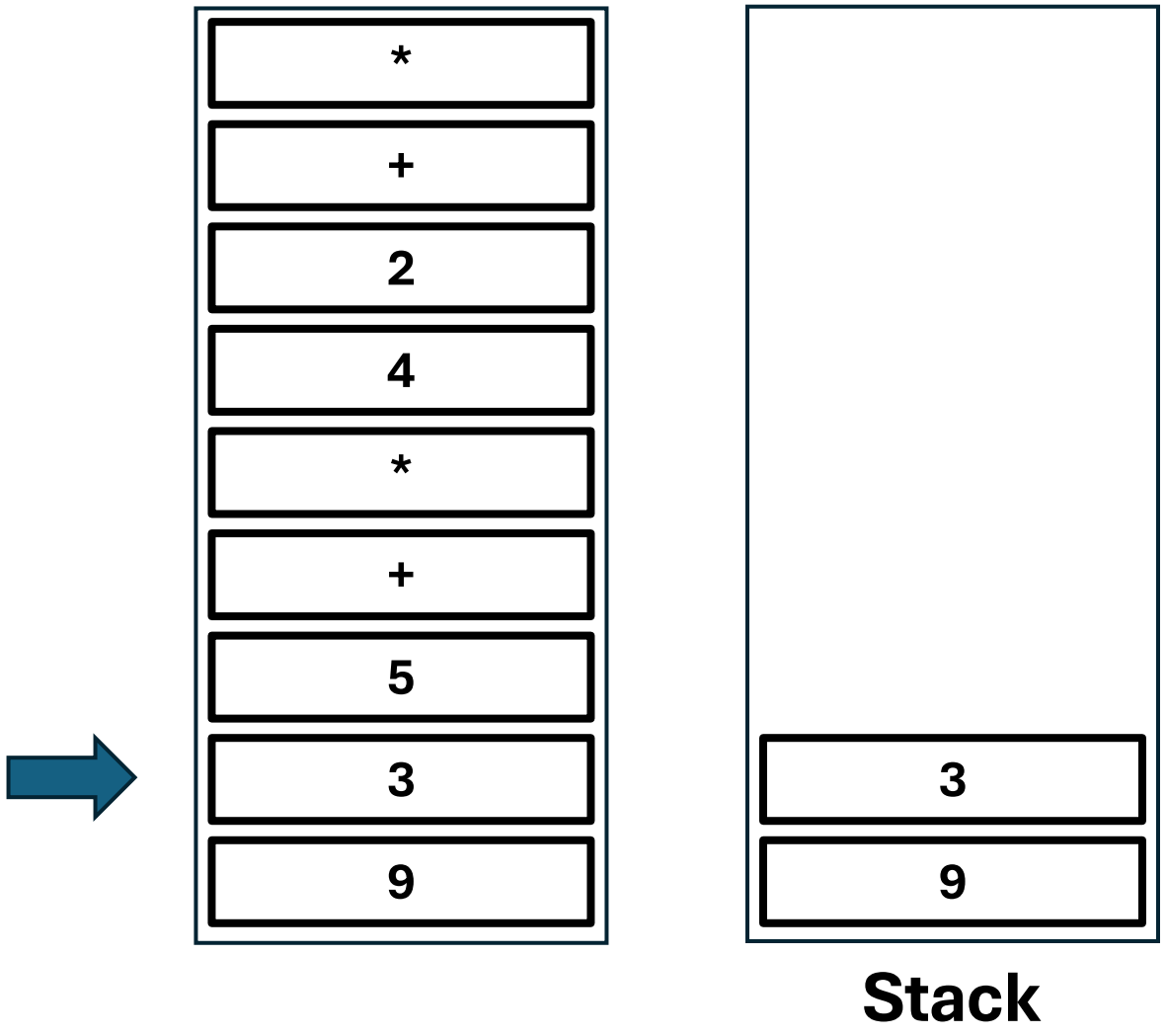
**→ Number**

**→ push to stack**

**Stack: [9]**

# Infix to Postfix expression

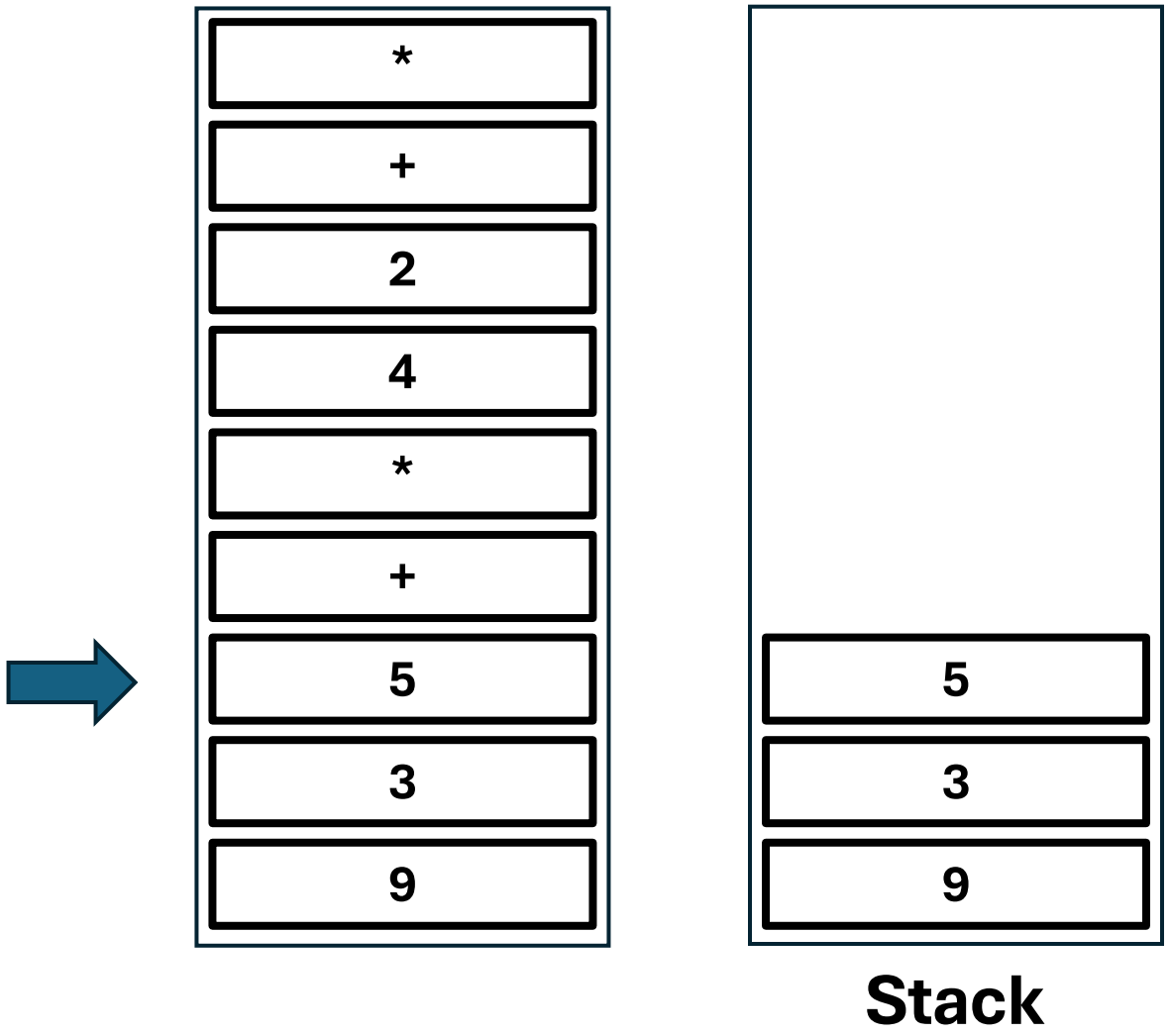
**[9, 3, 5, +, *, 4, 2, +, *]**




**Token: 3**
**→ Number**
**→ push to stack**
**Stack: [9, 3]**

# Infix to Postfix expression

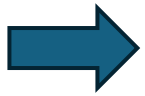
**[9, 3, 5, +, *, 4, 2, +, *]**

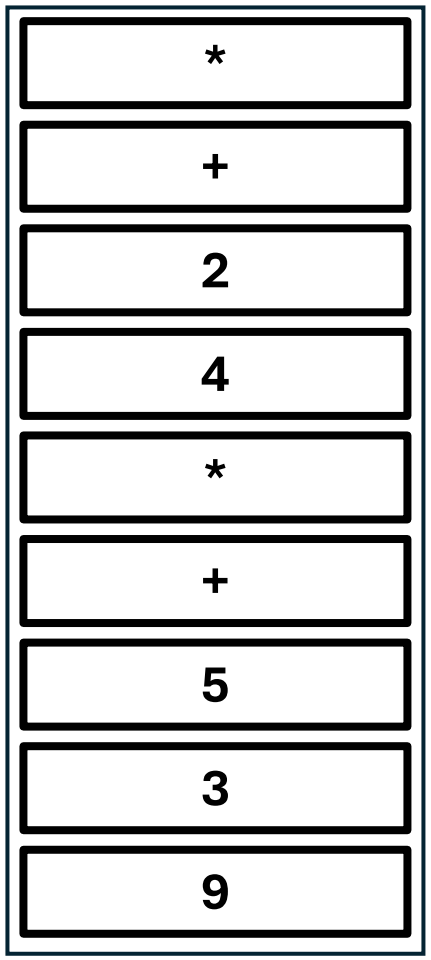


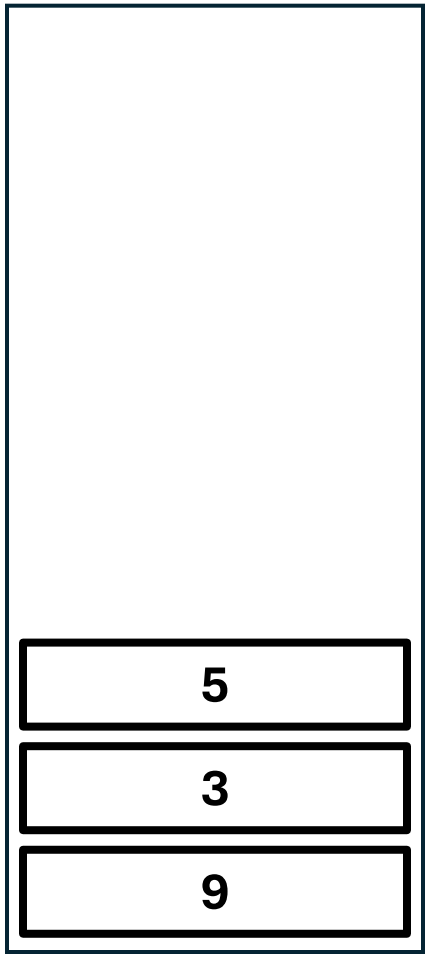

**Token: 5**
**→ Number**
**→ push to stack**
**Stack: [9, 3, 5]**

# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**

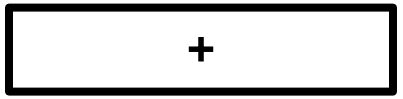




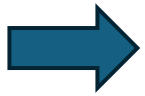

**Stack**

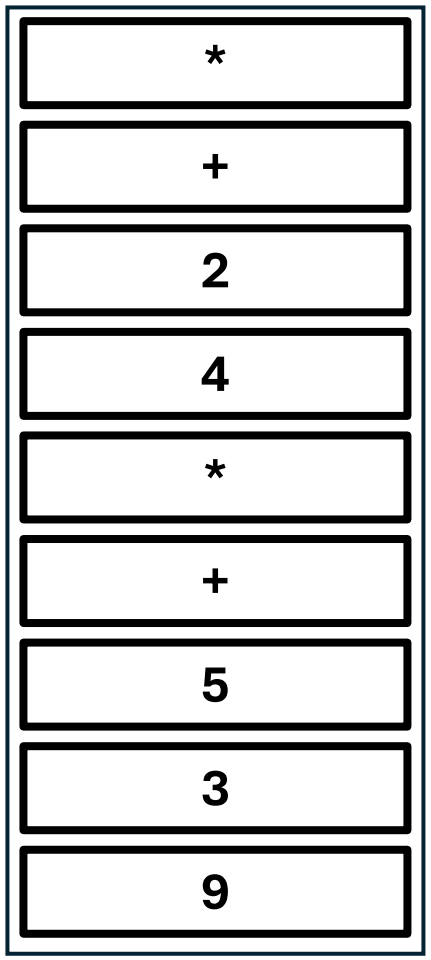
**Token: +**
**→ Operator**
**→ pop two: 5 and 3**
**→ Compute 3 + 5 = 8**
**→ push result**
**Stack: [9, 8]**

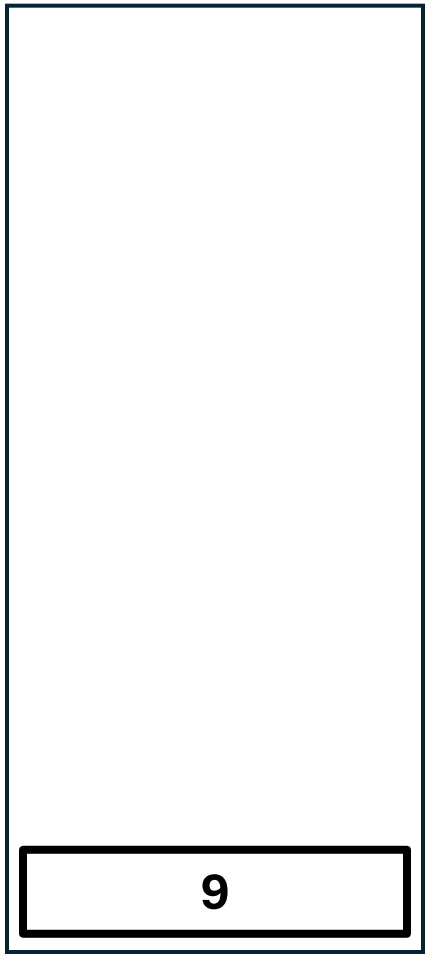


# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**

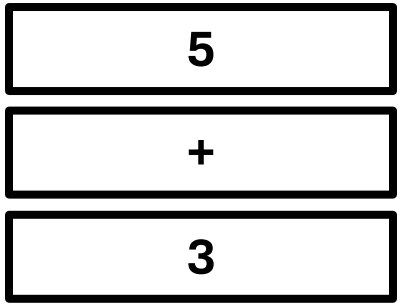




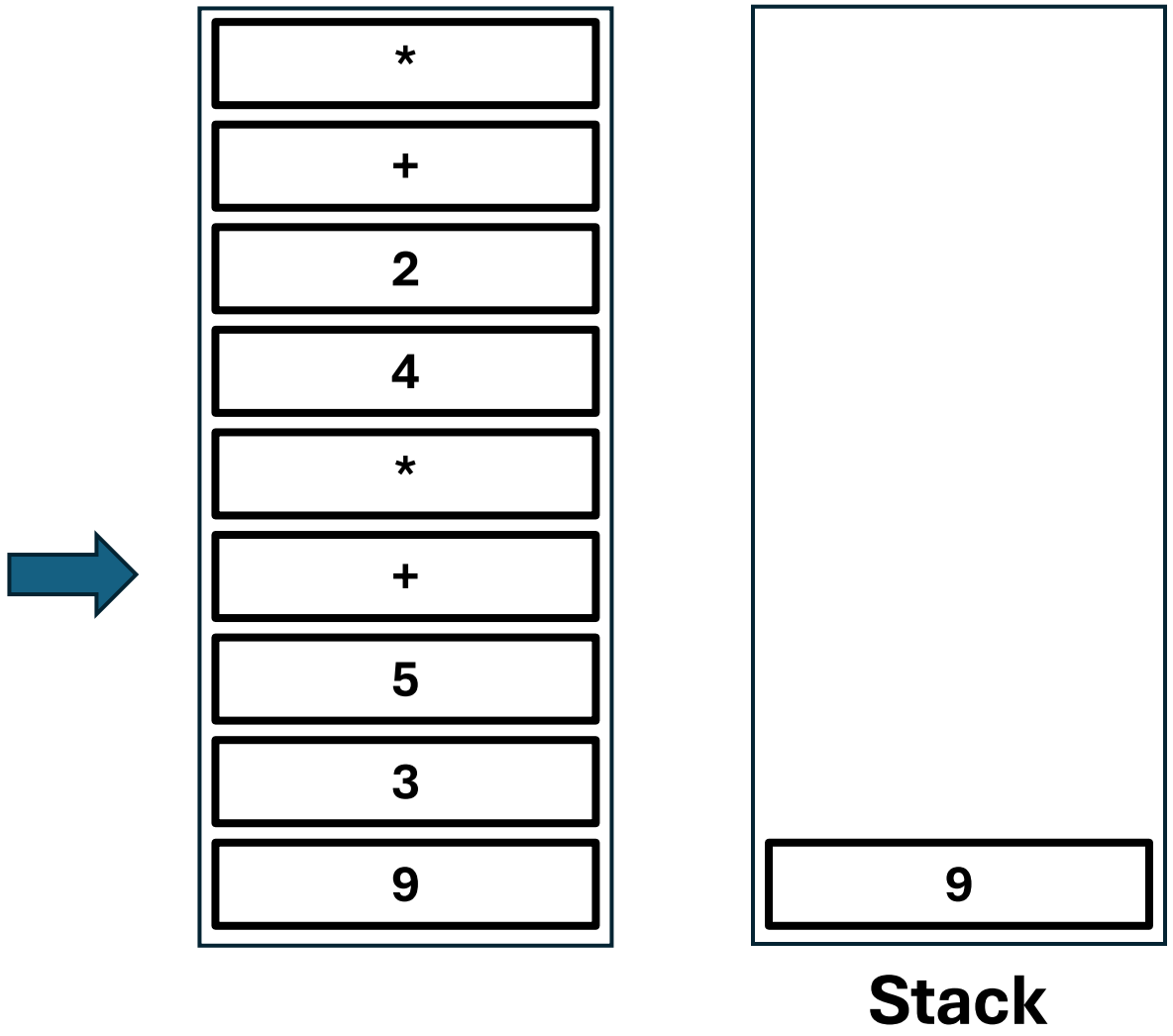

**Stack**

**Token: +**
**→ Operator**
**→ pop two: 5 and 3**
**→ Compute 3 + 5 = 8**
**→ push result**
**Stack: [9, 8]**



# Infix to Postfix expression

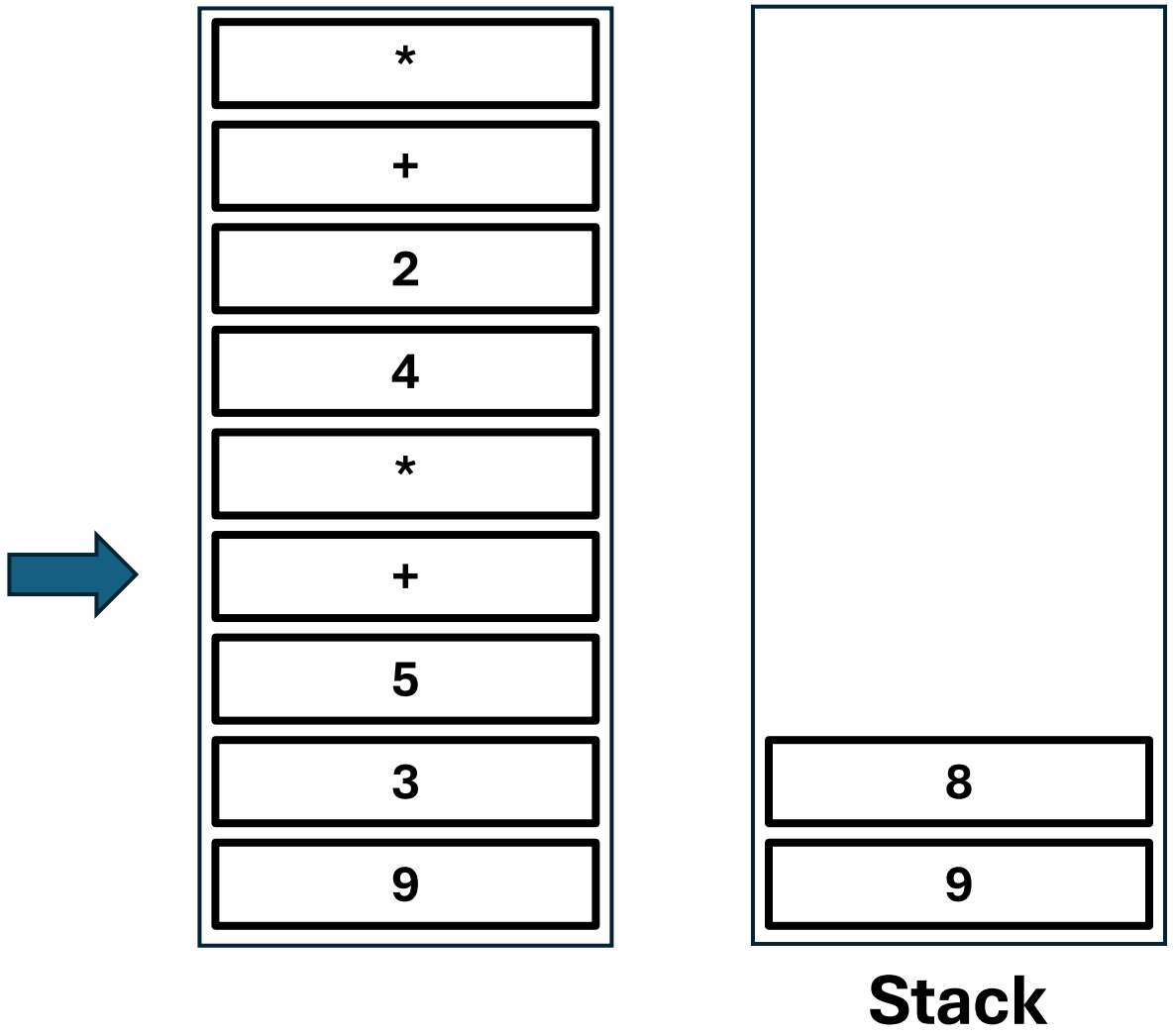
**[9, 3, 5, +, *, 4, 2, +, *]**




**Token: +**
**→ Operator**
**→ pop two: 5 and 3**
**→ Compute 3 + 5 = 8**
**→ push result**
**Stack: [9, 8]**

# Infix to Postfix expression

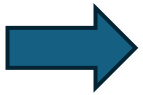
**[9, 3, 5, +, *, 4, 2, +, *]**

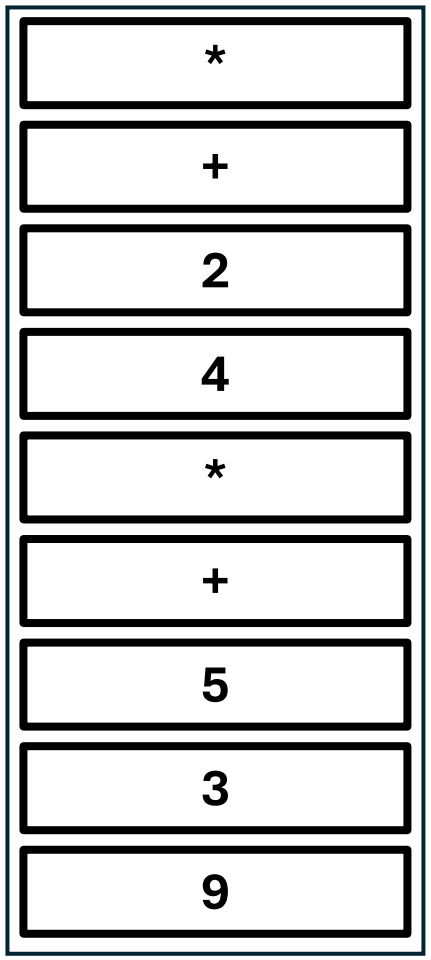


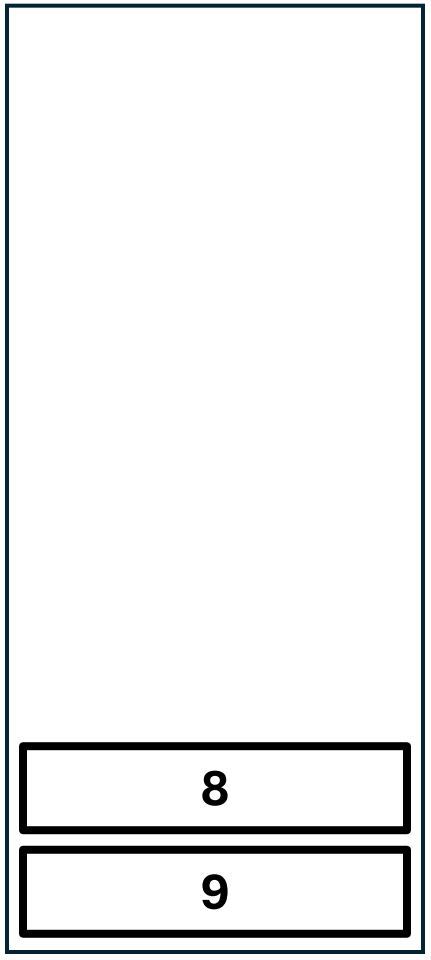

**Token: +**
**→ Operator**
**→ pop two: 5 and 3**
**→ Compute 3 + 5 = 8**
**→ push result**
**Stack: [9, 8]**

# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**

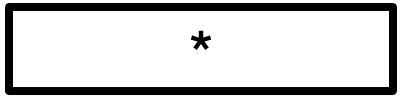




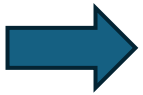

**Stack**

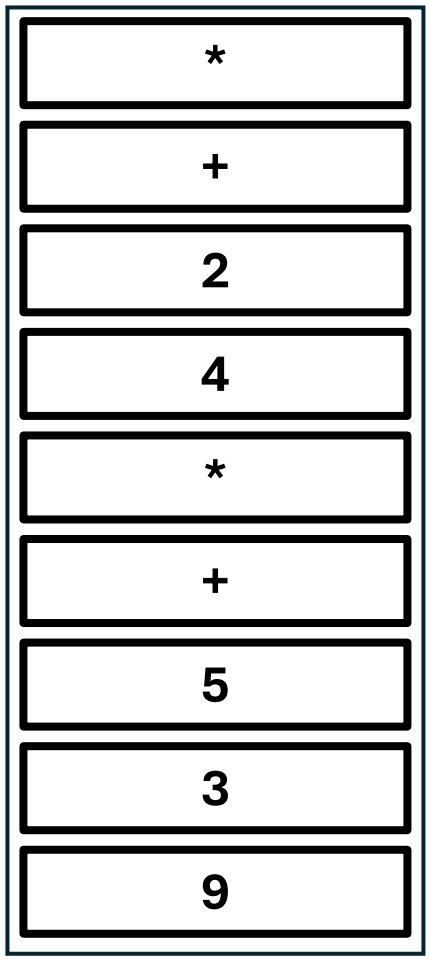
**Token: ***
**→ Operator**
**→ pop two: 8 and 9**
**→ Compute 9 * 8 = 72**
**→ push result**
**Stack: [72]**

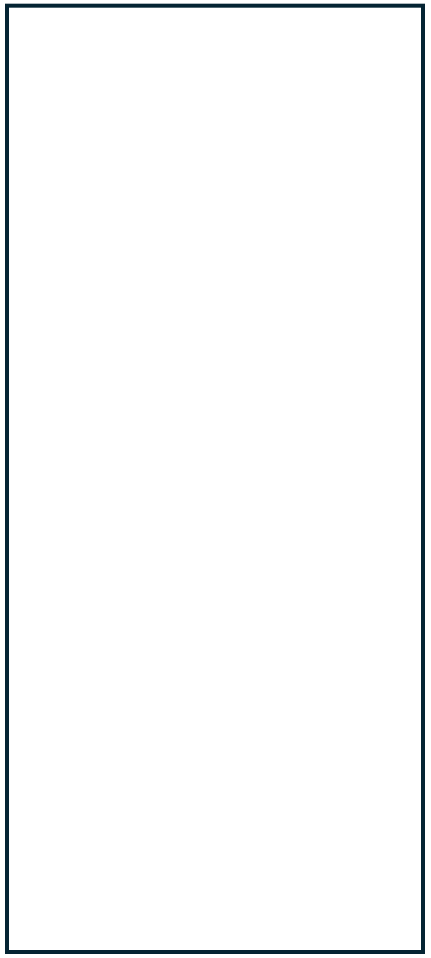


# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**

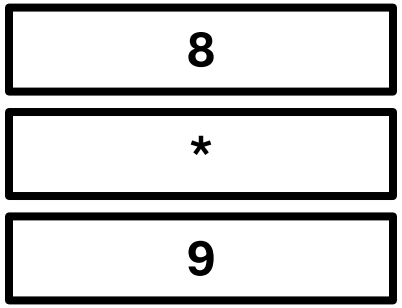




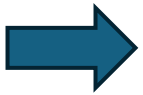
**Stack**

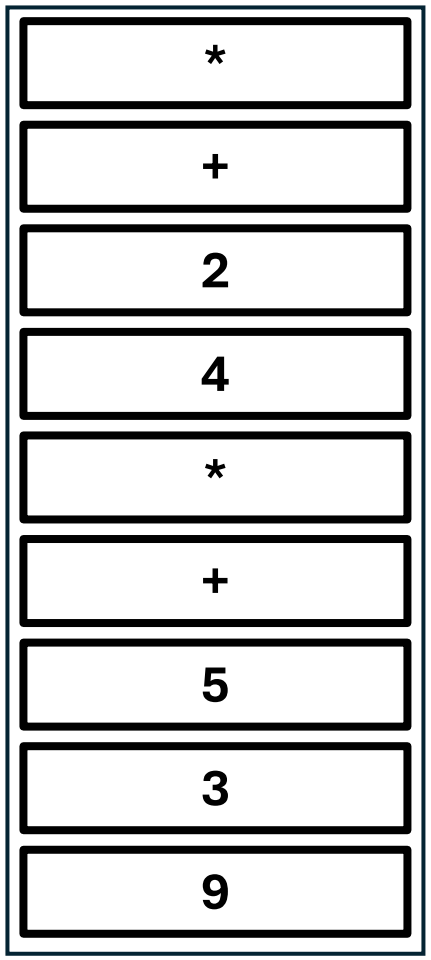
**Token: ***
**→ Operator**
**→ pop two: 8 and 9**
**→ Compute 9 * 8 = 72**
**→ push result**
**Stack: [72]**

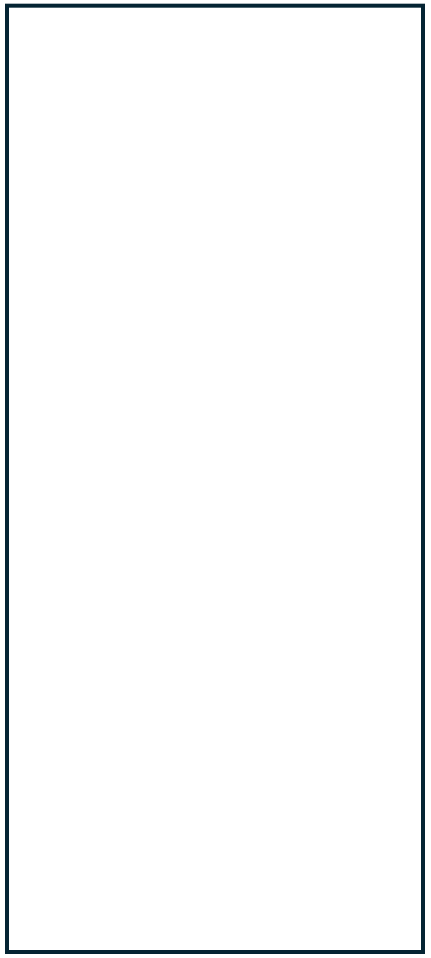


# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**





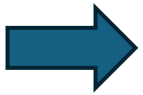



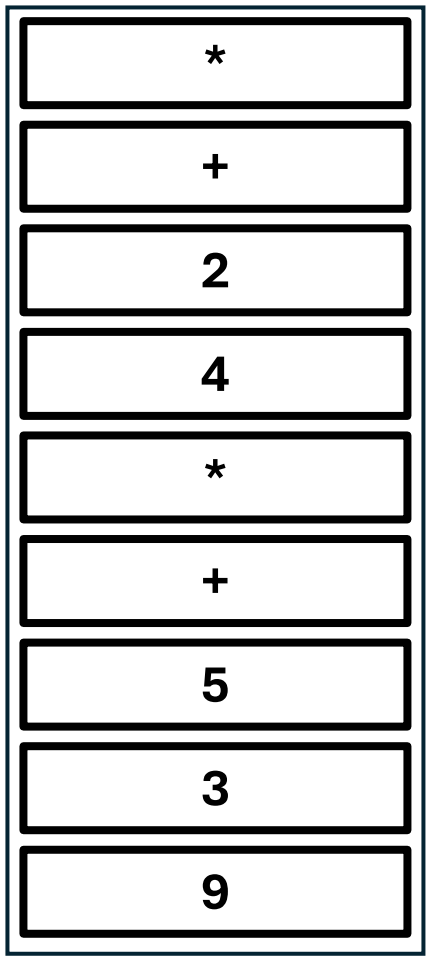
**Stack**

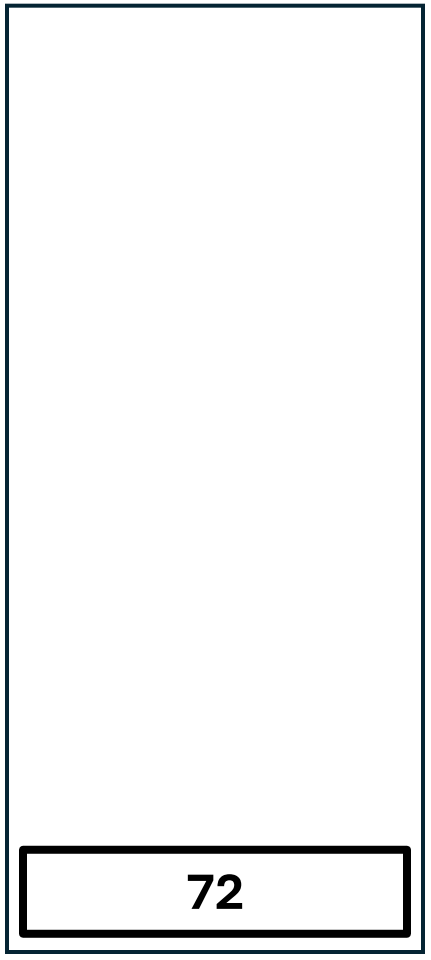
**Token: ***
**→ Operator**
**→ pop two: 8 and 9**
**→ Compute 9 * 8 = 72**
**→ push result**
**Stack: [72]**

# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**





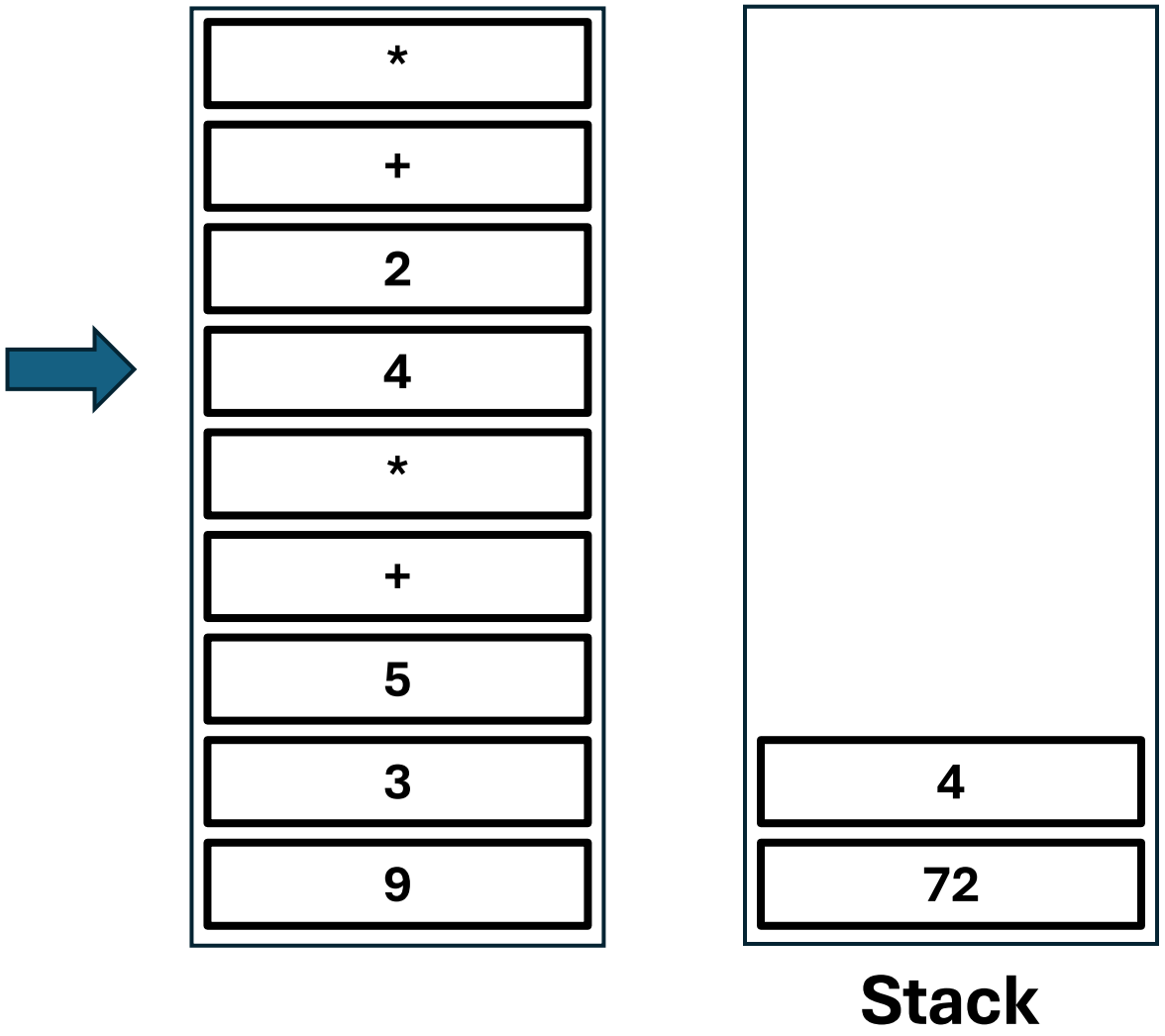




**Stack**

**Token: ***
**→ Operator**
**→ pop two: 8 and 9**
**→ Compute 9 * 8 = 72**
**→ push result**
**Stack: [72]**

# Infix to Postfix expression

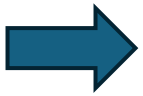
**[9, 3, 5, +, *, 4, 2, +, *]**

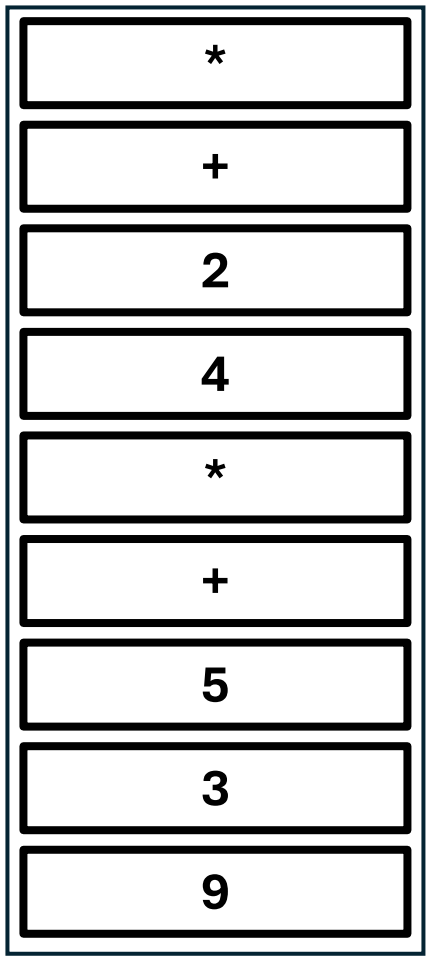


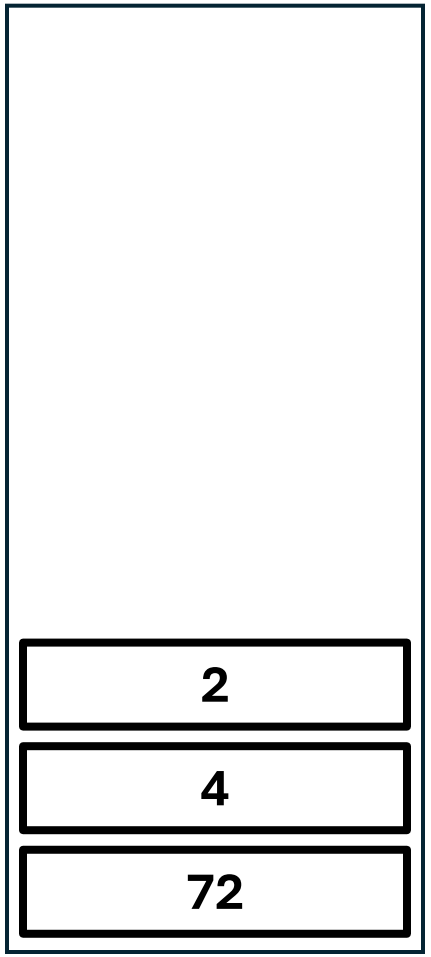

**Token: 4**
**→ Number**
**→ push**
**Stack: [72, 4]**

# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**





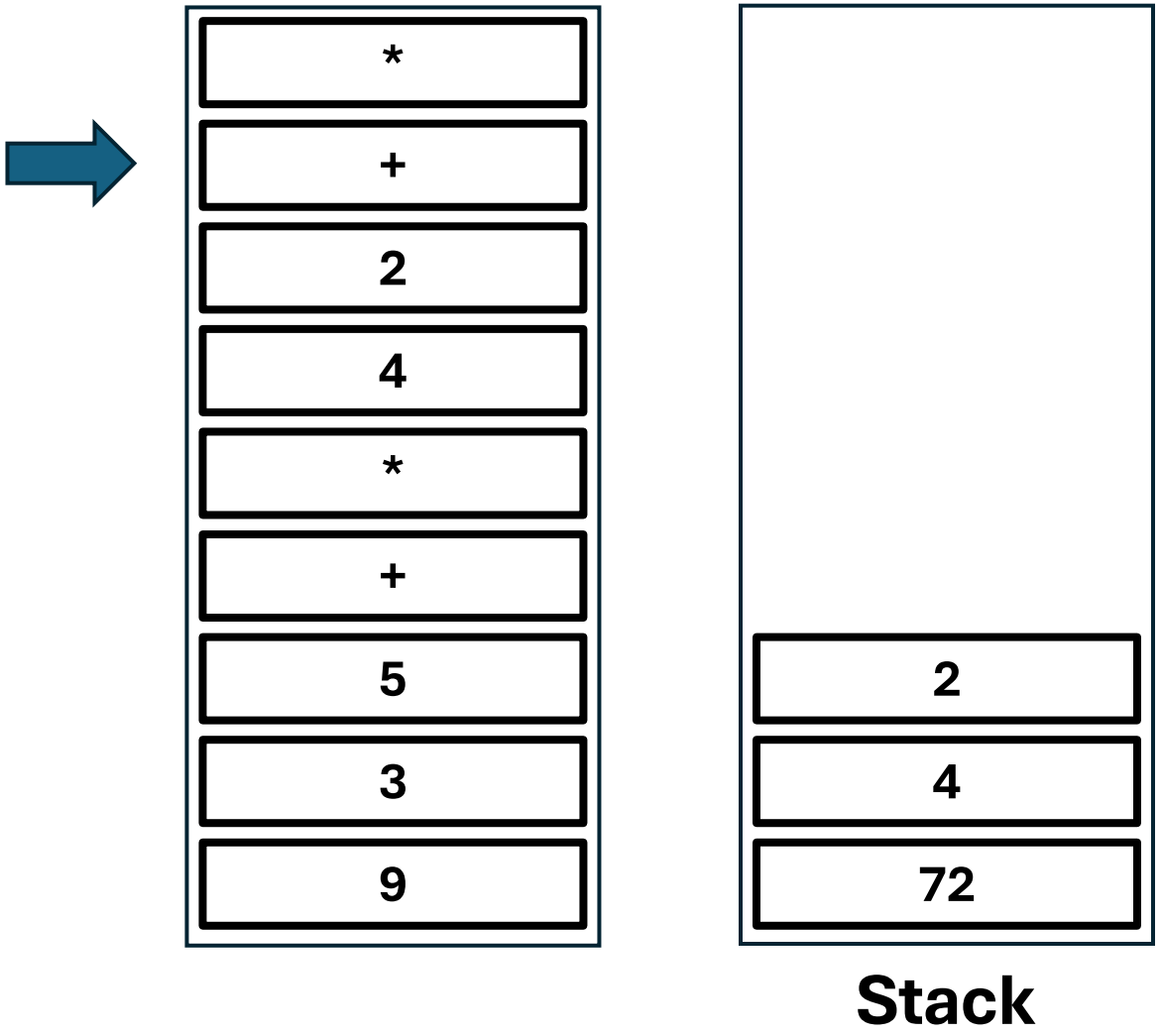




**Stack**

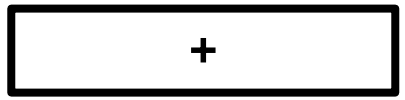
**Token: 2**
**→ Number**
**→ push**
**Stack: [72, 4, 2]**

# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**

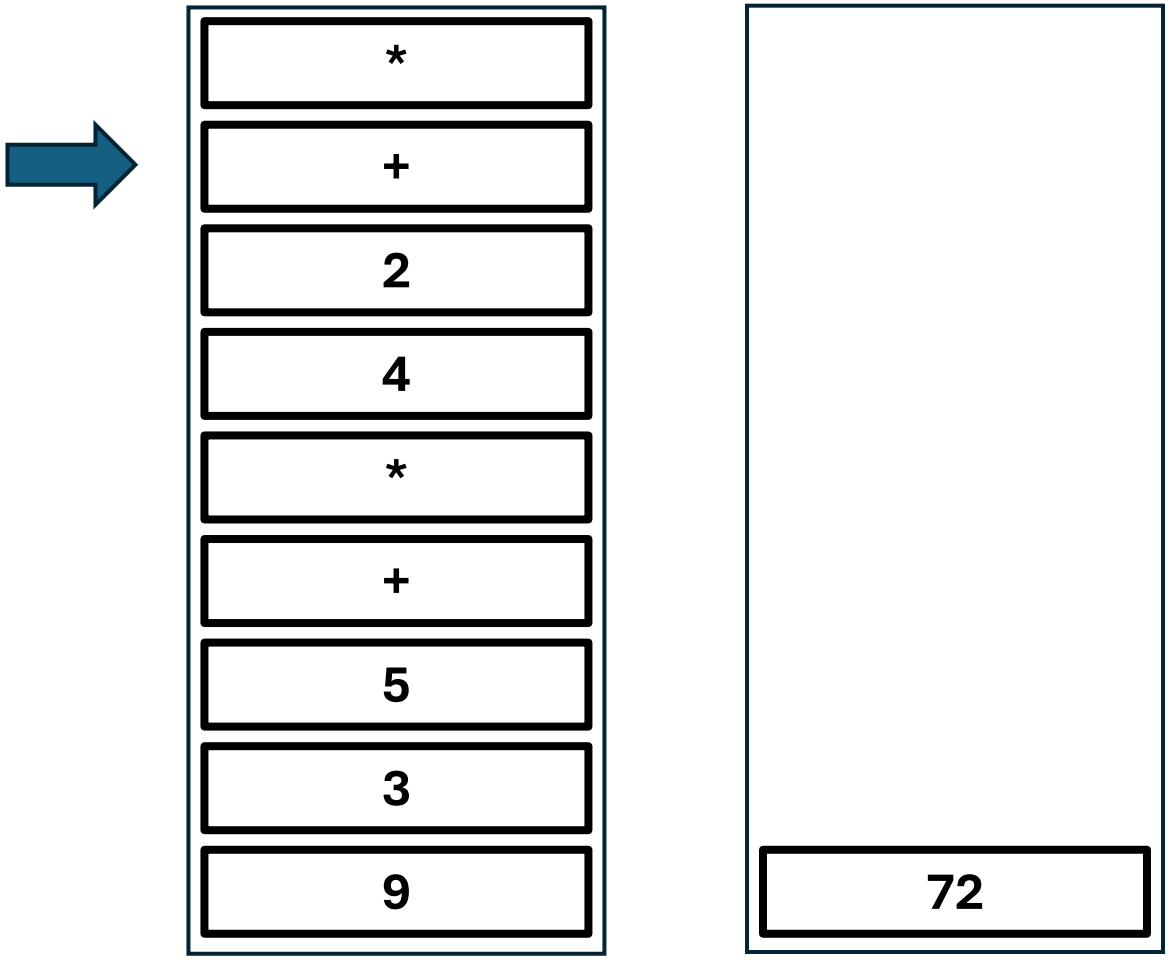



**Token: +**
**→ Operator**
**→ pop two: 2 and 4**
**→ Compute 4 + 2 = 6**
**→ push result**
**Stack: [72, 6]**

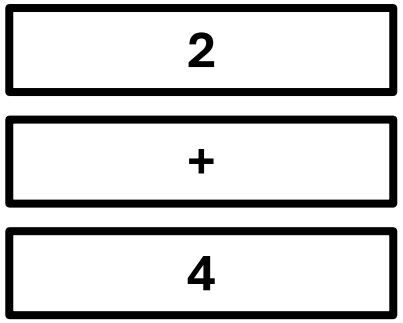


# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**

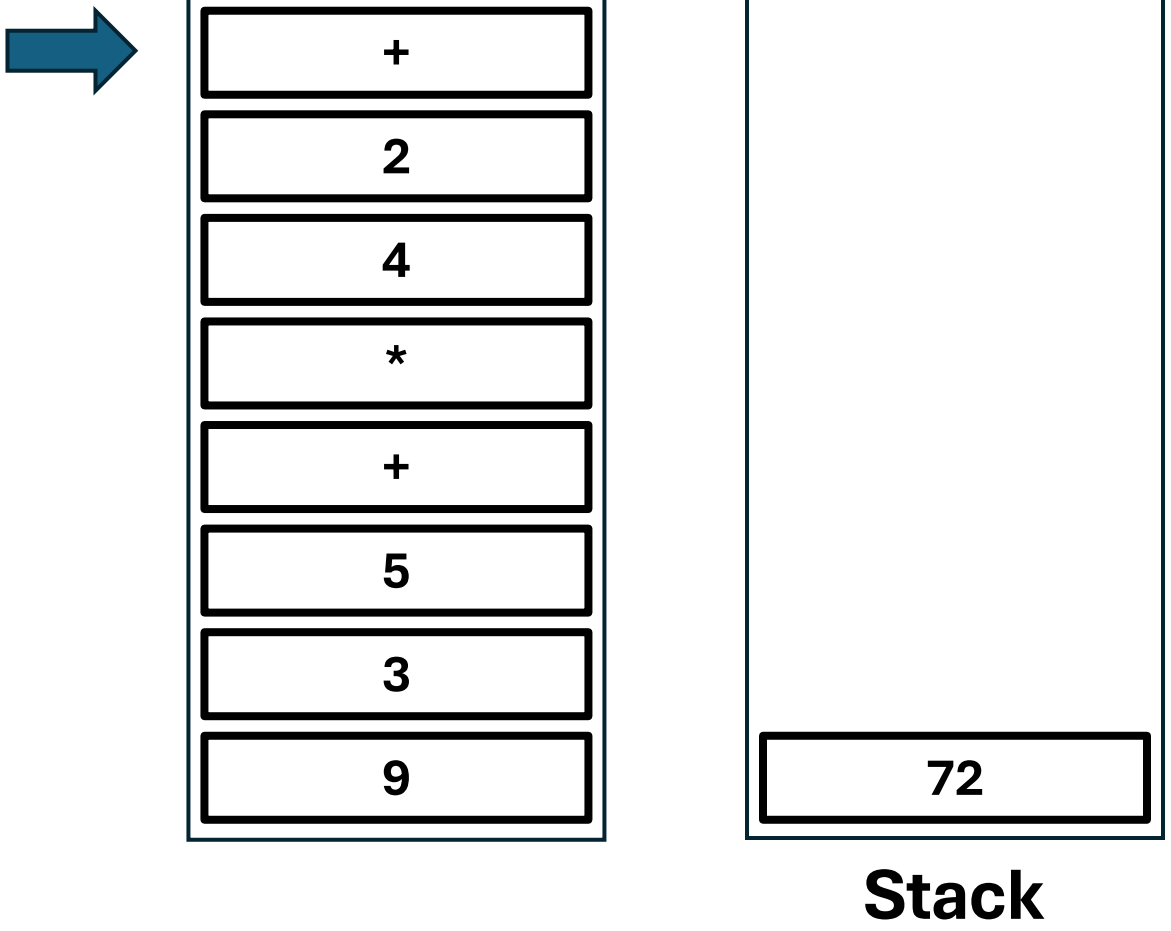



**Token: +**
**→ Operator**
**→ pop two: 2 and 4**
**→ Compute 4 + 2 = 6**
**→ push result**
**Stack: [72, 6]**

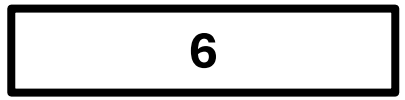


# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**

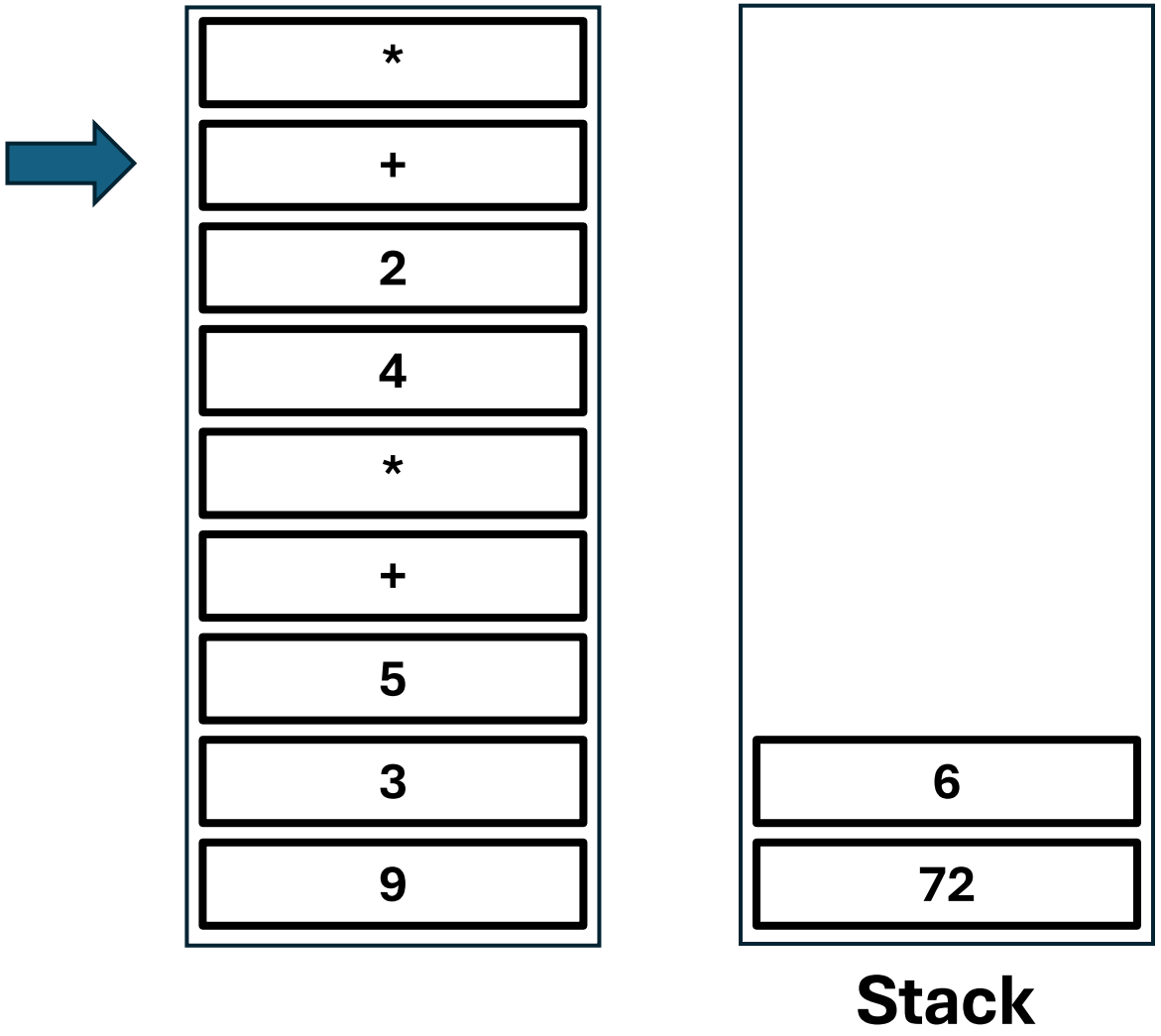



**Token: +**
**→ Operator**
**→ pop two: 2 and 4**
**→ Compute 4 + 2 = 6**
**→ push result**
**Stack: [72, 6]**



# Infix to Postfix expression

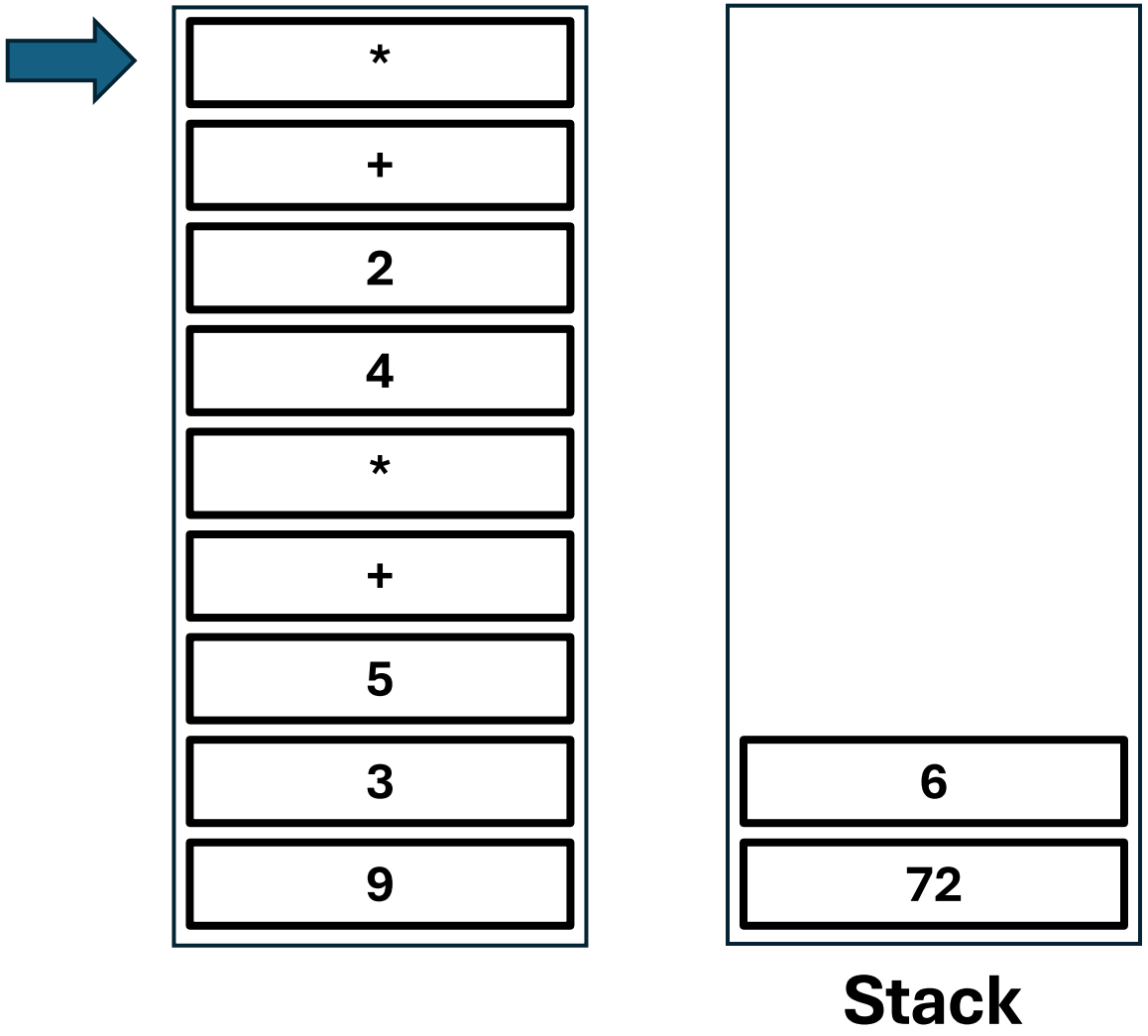
**[9, 3, 5, +, *, 4, 2, +, *]**



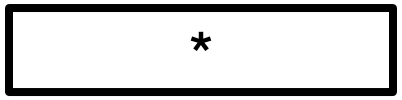

**Token: +**
**→ Operator**
**→ pop two: 2 and 4**
**→ Compute 4 + 2 = 6**
**→ push result**
**Stack: [72, 6]**

# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**

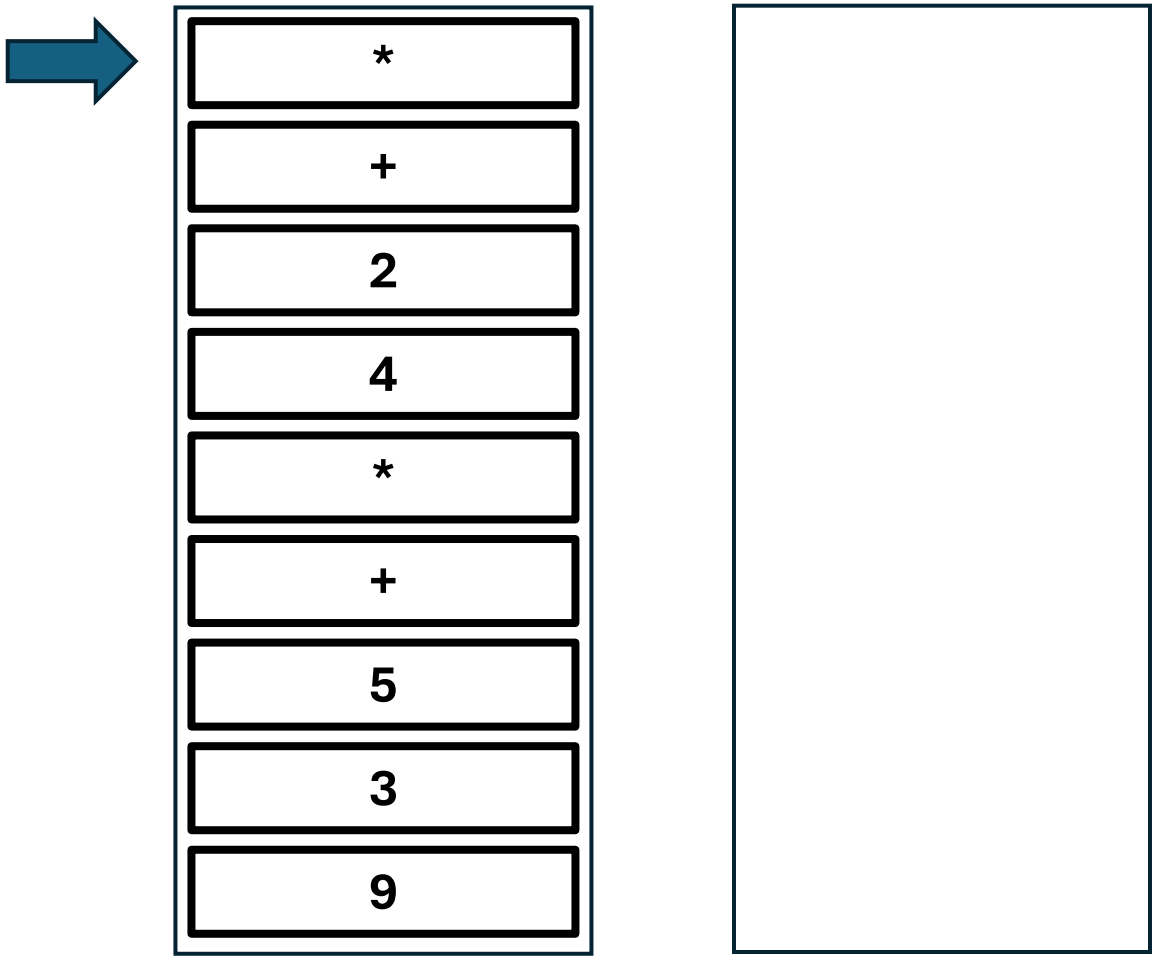



**Token: ***
**→ Operator**
**→ pop two: 6 and 72**
**→ Compute 72 * 6 = 432**
**→ push result**
**Stack: [432]**



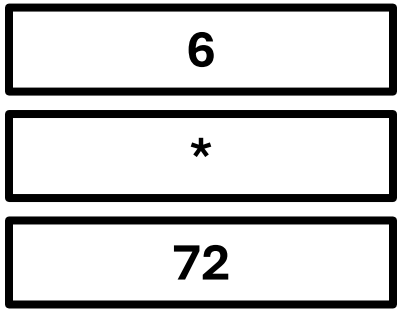
# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**



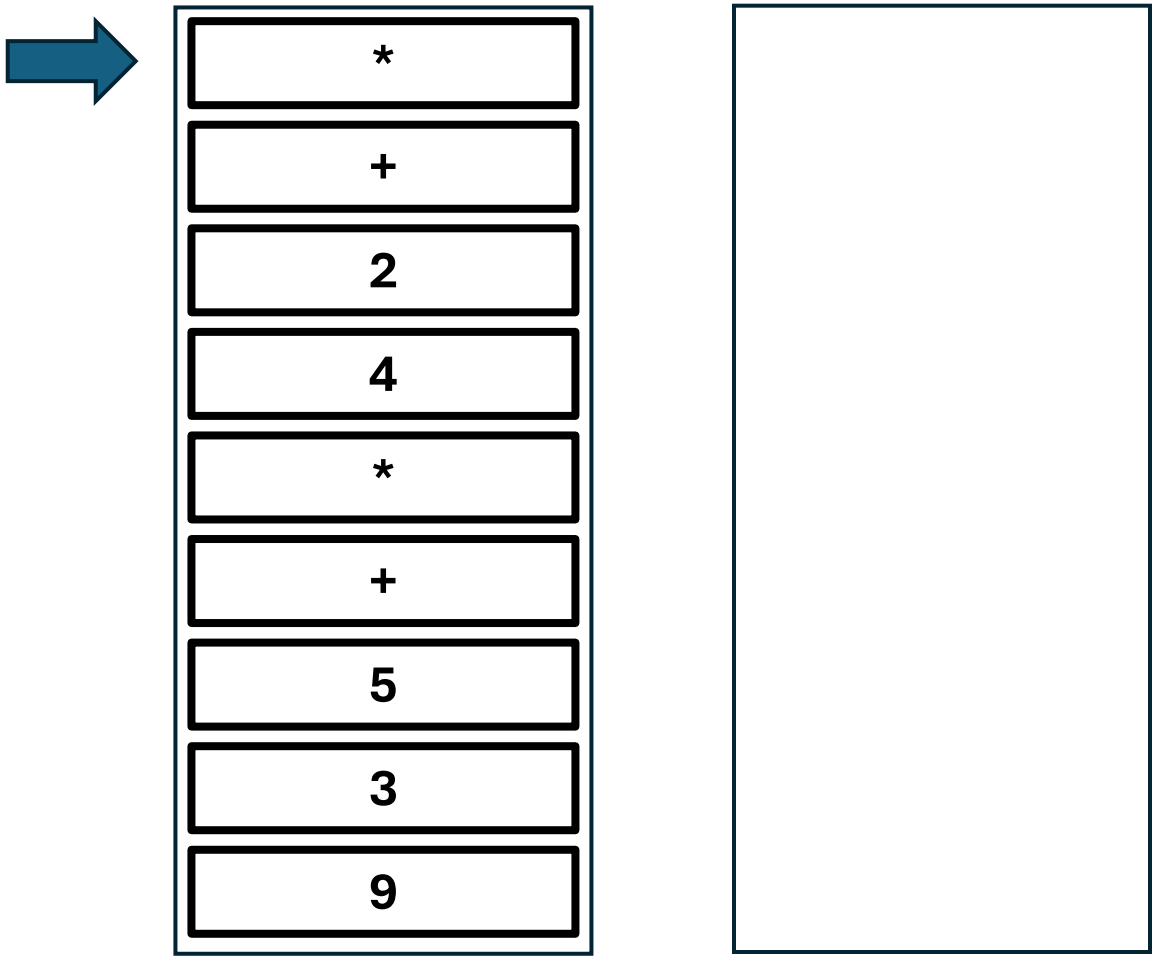

**Stack**

**Token: ***
**→ Operator**
**→ pop two: 6 and 72**
**→ Compute 72 * 6 = 432**
**→ push result**
**Stack: [432]**



# Infix to Postfix expression

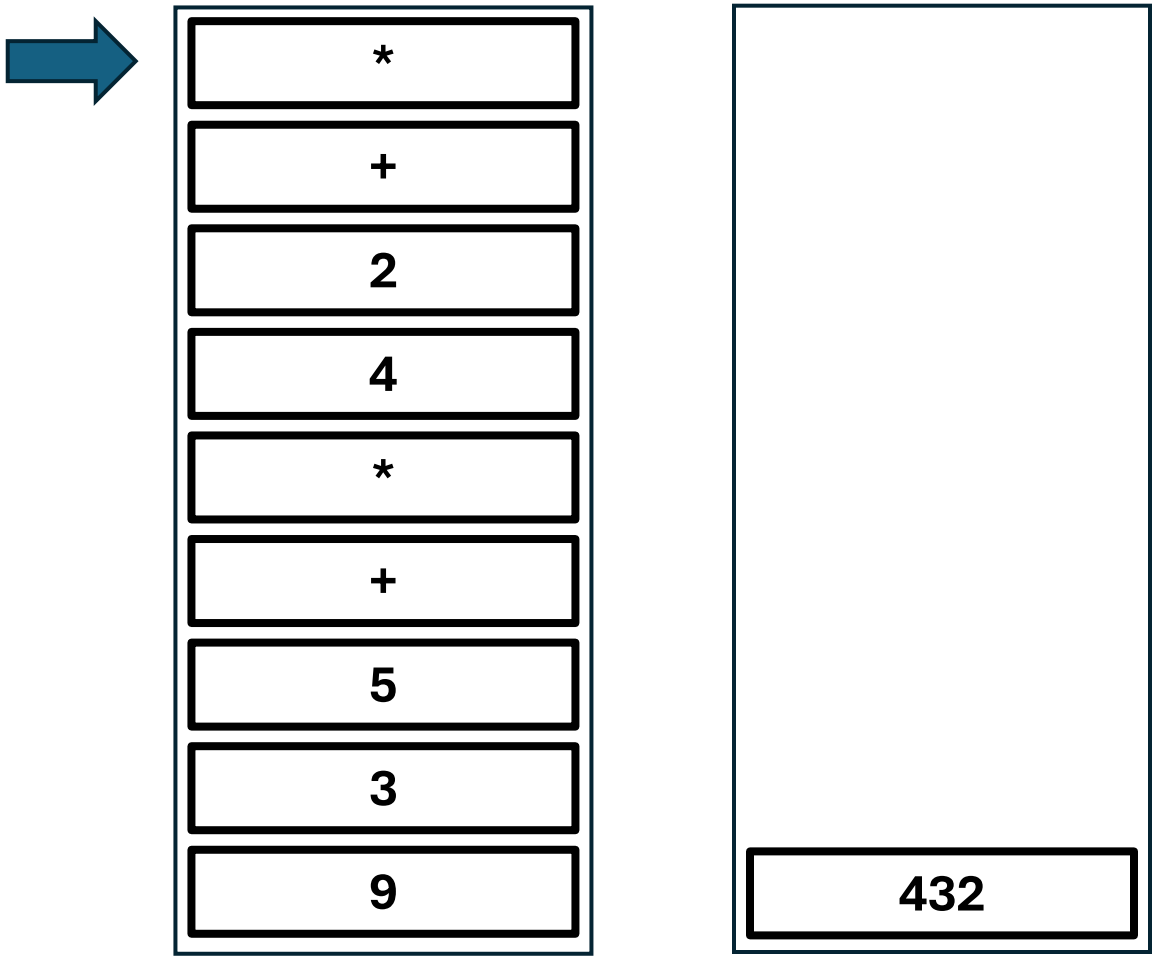
**[9, 3, 5, +, *, 4, 2, +, *]**




**Token: ***
**→ Operator**
**→ pop two: 6 and 72**
**→ Compute 72 * 6 = 432**
**→ push result**
**Stack: [432]**

# Infix to Postfix expression

**[9, 3, 5, +, *, 4, 2, +, *]**




**Token: ***
**→ Operator**
**→ pop two: 6 and 72**
**→ Compute 72 * 6 = 432**
**→ push result**
**Stack: [432]**


Get Full Source Code
of Convert any INFIX
to POSTFIX and
calculate it

# Include Header

```
#include <iostream>
#include <stack>
#include <sstream>
#include <cctype>
#include <string>
#include <map>
```

```cpp
using namespace std;

// Function to define operator precedence
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}

// Function to convert infix to postfix
string infixToPostfix(const string& infix) {
    string postfix = "";
    stack<char> s;

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

        if (isspace(token)) continue;

        if (isdigit(token)) {
            // Handle multi-digit numbers
            while (i < infix.length() && isdigit(infix[i])) {
                postfix += infix[i++];
            }
            postfix += " ";
            i--; // step back after loop
        }
        else if (token == '(') {
            s.push(token);
        }
        else if (token == ')') {
            while (!s.empty() && s.top() != '(') {
                postfix += s.top();
                postfix += " ";
                s.pop();
            }
            if (!s.empty()) s.pop(); // pop the '('
        }
        else { // operator
            while (!s.empty() && precedence(s.top()) >= precedence(token)) {
                postfix += s.top();
                postfix += " ";
                s.pop();
            }
            s.push(token);
        }
    }

    while (!s.empty()) {
        postfix += s.top();
        postfix += " ";
        s.pop();
    }

    return postfix;
}
```

```cpp
// Function to evaluate postfix expression
int evaluatePostfix(const string& postfix) {
    stack<int> s;
    stringstream ss(postfix);
    string token;

    while (ss >> token) {
        if (isdigit(token[0])) {
            s.push(stoi(token));
        }
        else {
            int b = s.top(); s.pop();
            int a = s.top(); s.pop();

            if (token == "+") s.push(a + b);
            else if (token == "-") s.push(a - b);
            else if (token == "*") s.push(a * b);
            else if (token == "/") s.push(a / b);
        }
    }

    return s.top();
}

// Main function
int main() {
    string infix;

    cout << "Enter an infix expression (e.g. 9 * (3 + 5) * (4 + 2)): ";
    getline(cin, infix);

    string postfix = infixToPostfix(infix);
    cout << "Postfix: " << postfix << endl;

    int result = evaluatePostfix(postfix);
    cout << "Result: " << result << endl;

    return 0;
}
```

# Jum tengok code conversion

```
1.  // Function to convert infix to postfix
2.  string infixToPostfix(const string& infix) {
3.      string postfix = "";
4.      stack<char> s;

5.      for (int i = 0; i < infix.length(); ++i) {
6.          char token = infix[i];

7.          if (isspace(token)) continue;

8.          if (isdigit(token)) {
9.              // Handle multi-digit numbers
10.             while (i < infix.length() && isdigit(infix[i])) {
11.                 postfix += infix[i++];
12.             }
13.             postfix += " ";
14.             i--; // step back after loop
15.         }
16.         else if (token == '(') {
17.             s.push(token);
18.         }
19.         else if (token == ')') {
20.             while (!s.empty() && s.top() != '(') {
21.                 postfix += s.top();
22.                 postfix += " ";
23.                 s.pop();
24.             }
25.             if (!s.empty()) s.pop(); // pop the '('
26.         }
27.         else { // operator
28.             while (!s.empty() && precedence(s.top()) >= precedence(token)) {
29.                 postfix += s.top();
30.                 postfix += " ";
31.                 s.pop();
32.             }
33.             s.push(token);
34.         }
35.     }

36.     while (!s.empty()) {
37.         postfix += s.top();
38.         postfix += " ";
39.         s.pop();
40.     }

41.     return postfix;
42. }
```

```
string infixToPostfix(const string& infix)
```

→ Ni function yang ambil string infix (contoh: 3 + 5 * (2 - 1))

→ Output dia: postfix (contoh: 3 5 2 1 - * +)

Masa di main function:

```
int main() {
    string infix;

    cout << "Enter an infix expression (e.g. 9 * (3 + 5) * (4 + 2)): ";
    getline(cin, infix);

    string postfix = infixToPostfix(infix);
    cout << "Postfix: " << postfix << endl;

    int result = evaluatePostfix(postfix);
    cout << "Result: " << result << endl;

    return 0;
}
```

```
string postfix = "";
stack<char> s;
```

➡️ postfix = tempat kita simpan jawapan akhir dalam bentuk postfix
➡️ s = stack untuk simpan operator (macam +, *, (, etc.)

```
for (int i = 0; i < infix.length(); ++i) {
        char token = infix[i];
```

➡️ Loop satu-satu huruf/character dalam infix
➡️ token = huruf semasa kita tengah baca

Jadi, kita sekarang masuk loop panjang ni, kita start dengan posisi int i=0, maksudnya huruf pertama yang kita detect. Kita letak dalam satu variable char nama “token”.

```
if (isspace(token)) continue;
```

➡️ Kalau token tu space (kosong), kita skip je

Ingat kita masih dalam loop, ini barisan seterusnya. isspace() tu wujud dalam header `#include <cctype>`. Kena ada sebab nak detect whitespace sebab kadang-kadang hangpa masukkan tempat kosonglah, newline la etc. Kena ingat apa dia buat adalah:

`The isspace() function returns a non-zero value (equivalent to boolean true) if a character is a whitespace character. Whitespace characters are spaces, tabs and newline characters. - website: https://www.w3schools.com/c/ref_ctype_isspace.php`

```
if (isdigit(token)) {
            // Handle multi-digit numbers
            while (i < infix.length() && isdigit(infix[i])) {
                postfix += infix[i++];
            }
            postfix += " ";
            i--; // step back after loop
        }
```

➡️ Kalau token ialah nombor (0-9):

Kita masuk nombor tu ke dalam postfix
Kalau nombor tu ada **lebih dari 1 digit** (contoh: 12, 100), kita loop dan ambil semua digit.
isdigit() tu wujud dalam header #include <cctype>.

Lepas tu kita tambah space " " untuk pisahkan antara nombor dan operator

i-- tu sebab dalam while, i dah ter-advance ke depan

```
if (isdigit(token)) {
            // Handle multi-digit numbers
            while (i < infix.length() && isdigit(infix[i])) {
                postfix += infix[i++];
            }
            postfix += " ";
            i--; // step back after loop
        }
```

Contohnya begini, jika nombor dalam infix adalah nombor 85 dan i = 0, tapi token adalah nombor 8 (sebab baru baca nombor 8 dari 85). Jadi, while loop ni check yg i = 0 dan masih kurang dari panjang infix.length () DAN isdigit pula check nombor posisi pertama adalah 8.

Jika betul, dia setkan postfix string kita tu equal to postfix+infix[i++] (membawa nombor 5) tapi KHAS untuk while loop ni aje. Kemudian, dia masuk balik while loop ni dan check i yang dah jadi = 1 dan amik isdigit = 5. dan tambah ke postfix menjadikan nombor sekarang 85.

```
if (isdigit(token)) {
            // Handle multi-digit numbers
            while (i < infix.length() && isdigit(infix[i])) {
                postfix += infix[i++];
            }
            postfix += " ";
            i--; // step back after loop
        }
```

Contoh jika

infix = 100+20

| Step | i | infix[i] | isdigit(infix[i]) | postfix | Action |
|---|---|---|---|---|---|
| 1 | 0 | '1' | true | '1' | i++ → i = 1 |
| 2 | 1 | '0' | true | '10' | i++ → i = 2 |
| 3 | 2 | '0' | true | '100' | i++ → i = 3 |
| 4 | 3 | '+' | false | | Keluar while |
| 5 | 3 | '+' | - | '100 ' | postfix += " " |
| 6 | 3 | '+' | - | '100 ' | i-- → i = 2 |
| 7 | 2→3 | '+' | - | - | i++ next loop → i = 3 → operator processed here |

```
else if (token == '(') {
            s.push(token);
        }
```

➡️ Kalau jumpa ( → kita simpan dalam stack
➡️ Sebab apa? Sebab nanti kita akan cari ) dan tahu bila nak kira

```
else if (token == ')') {
            while (!s.empty() && s.top() != '(') {
                postfix += s.top();
                postfix += " ";
                s.pop();
            }
            if (!s.empty()) s.pop(); // pop the '('
        }
```

➡️ Bila jumpa ) :

Kita pop semua operator dari stack ke postfix sampai jumpa (
Lepas jumpa (, kita buang ( tu (jangan letak dalam postfix)

```
else { // operator
            while (!s.empty() && precedence(s.top()) >= precedence(token)) {
                postfix += s.top();
                postfix += " ";
                s.pop();
            }
            s.push(token);
        }
```

➡️ Kalau token ialah operator macam +, *, /:

Kita tengok kalau operator dalam stack ada **keutamaan sama atau lebih tinggi**, kita pop ke postfix

Lepas tu baru kita push operator sekarang ke stack

Contoh:* lebih kuat dari +, so * stay dalam stack lebih lama

```c
// Function to define operator precedence
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}
```

Jangan lupa kita ada satu function bertaraf int yang check operator precedence.

**Sebab antara + dan – dengan * dan /, darab dan bahagi lebih tinggi statusnya.**

```
else { // operator
            while (!s.empty() && precedence(s.top()) >= precedence(token)) {
                postfix += s.top();
                postfix += " ";
                s.pop();
            }
            s.push(token);
        }
```

So contohnya dalam stack ada *, tetapi yang terbaru nak dimasukkan ini adalah +,

Jadi apa yg berlaku dia popkan * tu ke postfix kemudian add space tu dan popkan dari stack.

Seterusnya dia masukkan + pula dalam stack

```
else { // operator
            while (!s.empty() && precedence(s.top()) >= precedence(token)) {
                postfix += s.top();
                postfix += " ";
                s.pop();
            }
            s.push(token);
        }
```

Sebaliknya, jika yang ada dalam stack tu + tapi yang nak masuk tu *, maka kita push terus ke stack dan JANGAN add to postfix lagi. Tunggu dulu.

```
while (!s.empty()) {
        postfix += s.top();
        postfix += " ";
        s.pop();
    }
```

➡️ Bila habis baca infix, kita pop semua operator dalam stack
→ masuk dalam postfix

# Jum tengok code conversion

```
1. // Function to evaluate postfix expression
2. int evaluatePostfix(const string& postfix) {
3.     stack<int> s;
4.     stringstream ss(postfix);
5.     string token;

6.     while (ss >> token) {
7.         if (isdigit(token[0])) {
8.             s.push(stoi(token));
9.         }
10.        else {
11.            int b = s.top(); s.pop();
12.            int a = s.top(); s.pop();

13.            if (token == "+") s.push(a + b);
14.            else if (token == "-") s.push(a - b);
15.            else if (token == "*") s.push(a * b);
16.            else if (token == "/") s.push(a / b);
17.        }
18.    }
19.    return s.top();
20.}
```

```
string infixToPostfix(const string& infix)
```

→ Ni function yang ambil string infix (contoh: 3 + 5 * (2 - 1))

→ Output dia: postfix (contoh: 3 5 2 1 - * +)

Masa di main function:

```
int main() {
    string infix;

    cout << "Enter an infix expression (e.g. 9 * (3 + 5) * (4 + 2)): ";
    getline(cin, infix);

    string postfix = infixToPostfix(infix);
    cout << "Postfix: " << postfix << endl;

    int result = evaluatePostfix(postfix);
    cout << "Result: " << result << endl;

    return 0;
}
```

```
int evaluatePostfix(const string& postfix)
```

➡️ Function ni terima postfix expression dalam bentuk string

➡️ Contoh: "3 4 5 * +“

➡️ Output: jawapan kira-kira dia (contoh: 3 + (4 * 5) = 23)

```
stack<int> s;
stringstream ss(postfix);
string token;
```

✅ stack<int> s;

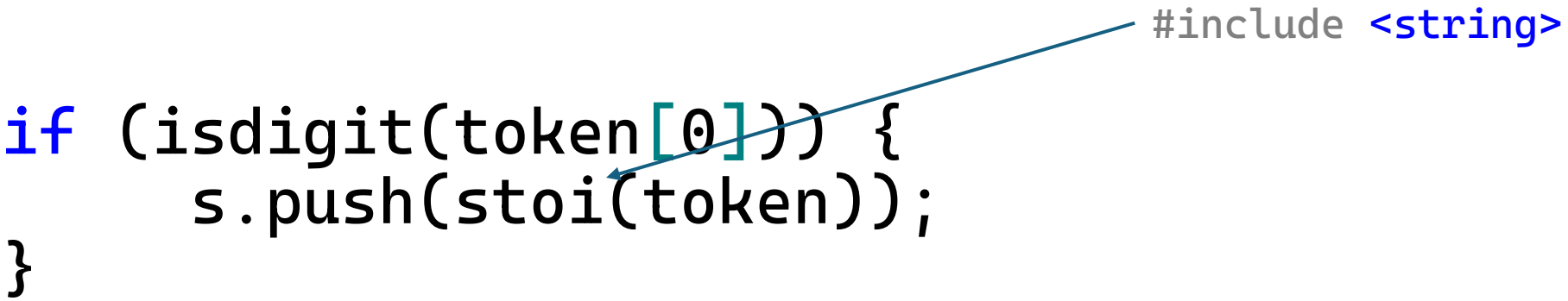
→ stack tempat kita simpan nombor sementara masa kira

✅ stringstream ss(postfix);

→ nak pisahkan postfix ikut space ("3 4 5 * +" → "3", "4", "5", "*", "+")

✅ token

→ untuk baca satu per satu item dari postfix




➡️ Kalau token tu nombor (contoh: "3", "45"):

Kita convert string ke nombor (stoi)
Dan masukkan dalam stack

✅ Contoh:Postfix = 3 4 +
→ Stack mula-mula: kosong
→ Baca 3 → masuk stack → [3]
→ Baca 4 → masuk stack → [3, 4]

```
else {
    int b = s.top(); s.pop();
    int a = s.top(); s.pop();
```

➡️ Kalau token BUKAN nombor (maksudnya operator macam +, -, *, /):

Ambil dua nombor paling atas dari stack (b & a)

Order penting: a dulu, baru b

→ contoh: kalau a = 3, b = 4, dan operator +,

kita kira 3 + 4

```
if (token == "+") s.push(a + b);
            else if (token == "-") s.push(a - b);
            else if (token == "*") s.push(a * b);
            else if (token == "/") s.push(a / b);
```

➡️ Lepas kira, masukkan result balik ke dalam stack

✅ Contoh:

Stack: [3, 4]

Token: "+“

Kira 3 + 4 = 7

→ Stack jadi [7]

```
return s.top();
```

➡️ Bila dah habis loop semua token, kita return nilai paling atas dalam stack → itulah jawapan akhir!

# Contoh Laluan Function

```
evaluatePostfix("3 4 5 * +")
```

➡️ Step:
Push 3 → [3]
Push 4 → [3, 4]
Push 5 → [3, 4, 5]

* → 4 * 5 = 20 → [3, 20]
+ → 3 + 20 = 23 → [23]

✅ Output: 23