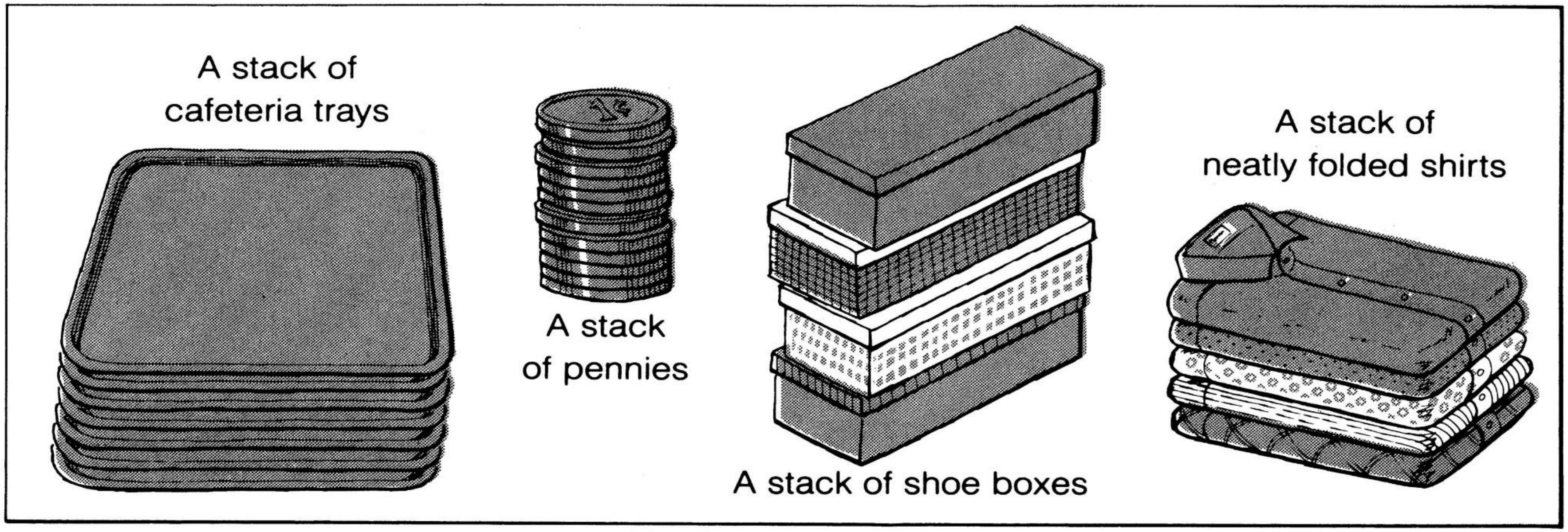
Stack

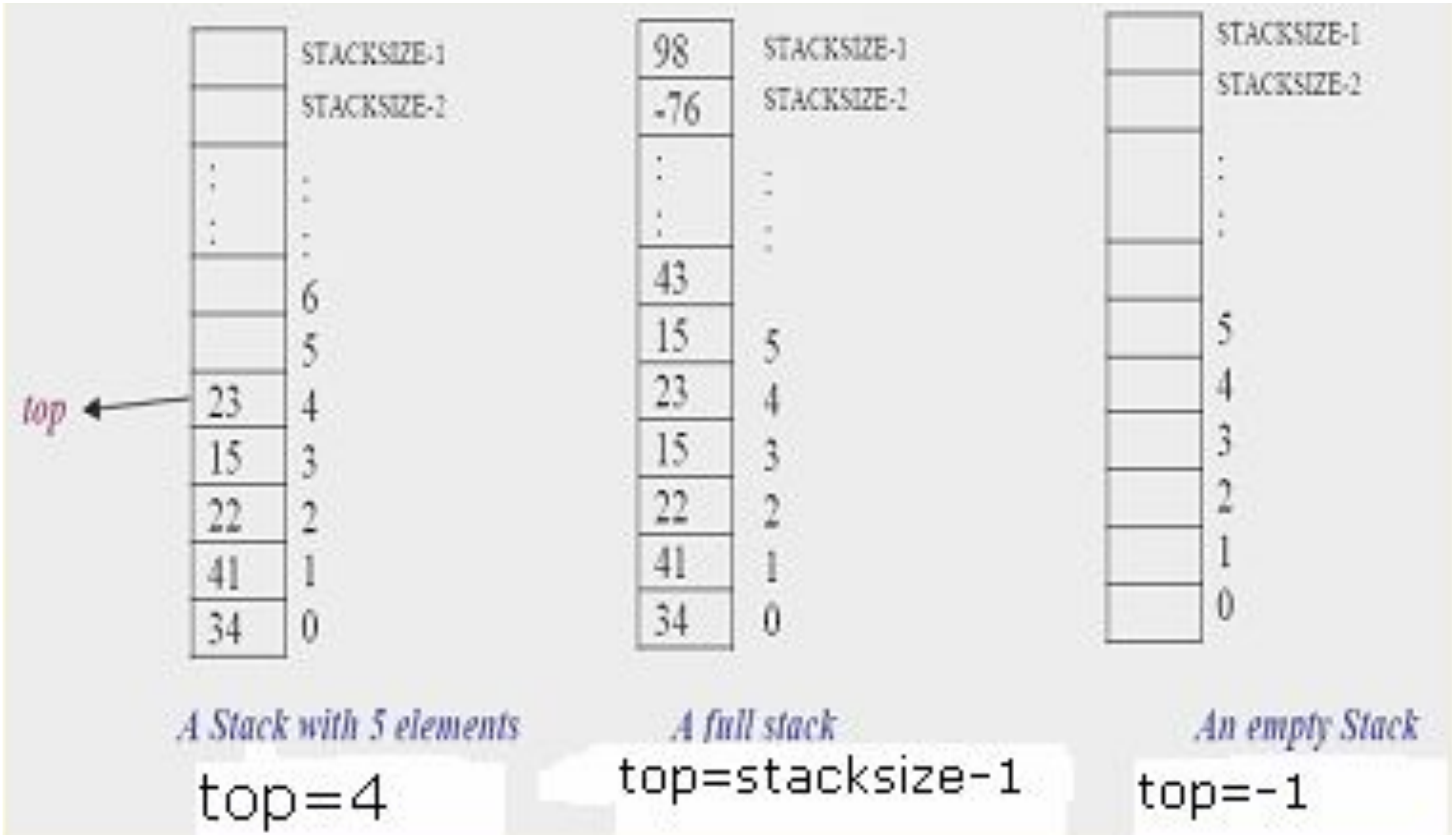
Qurrat-ul-ann

Stack

- A **Stack** is a **LIFO (Last In, First Out)** data structure where elements are added and removed from the **top**.
- The elements are removed in reverse order of that in which they were inserted into the stack.



Stack


STACKSIZE-1
STACKSIZE-2
6
5
top
23 4
15 3
22 2
41 1
34 0
A Stack with 5 elements
top=4
98 STACKSIZE-1
-76 STACKSIZE-2
43
15 5
23 4
15 3
22 2
41 1
34 0
A full stack
top=stacksize-1
STACKSIZE-1
STACKSIZE-2
5
4
3
2
1
0
An empty Stack
top=-1

Issues with stack

Stack overflow

The condition resulting from trying to push an element onto a full stack.

```
if(!stack.IsFull())
     stack.Push(item);
```

Stack underflow

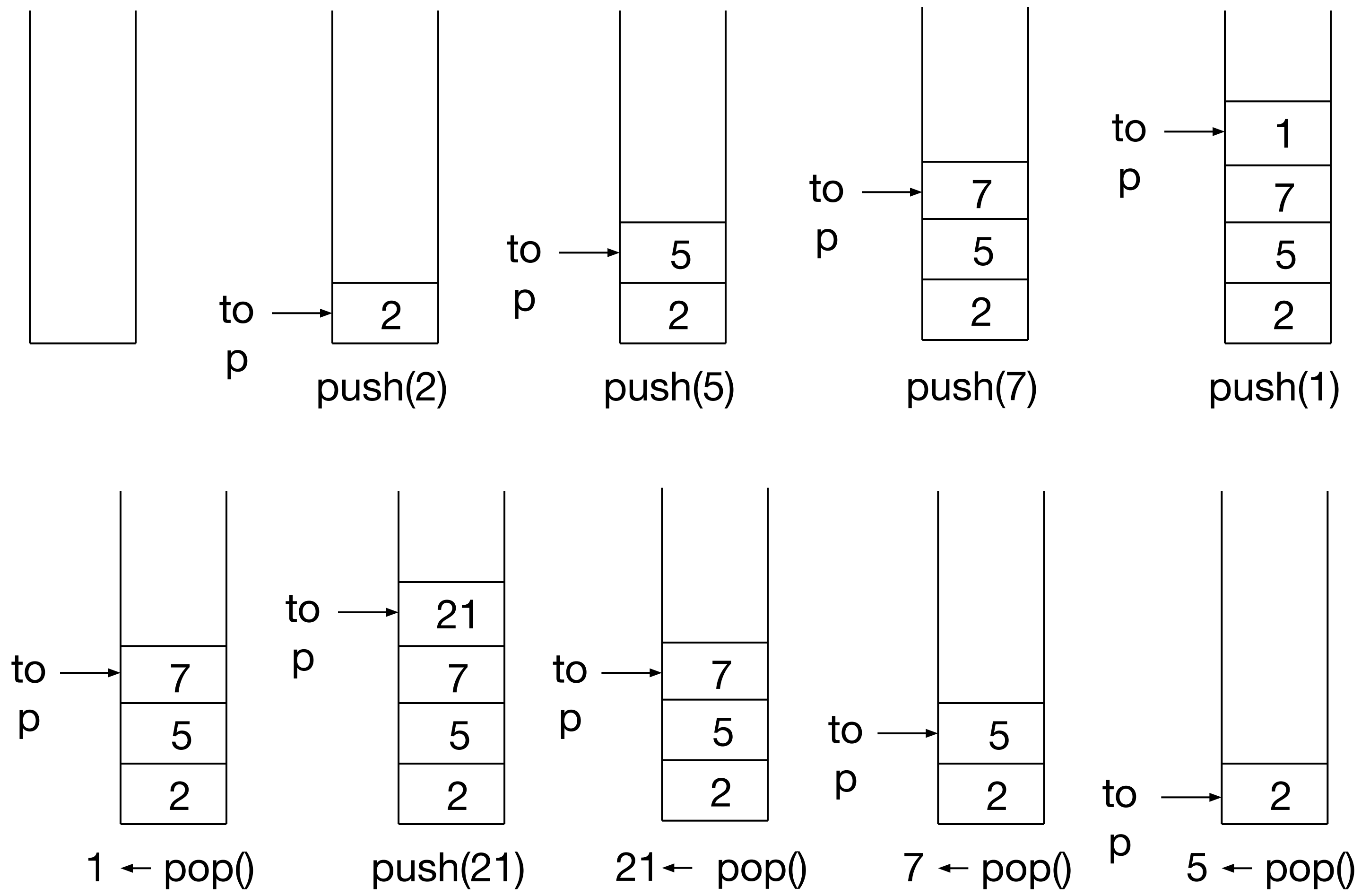
The condition resulting from trying to pop an empty stack.

```
if(!stack.IsEmpty())
     stack.Pop(item);
```

Stack Operations

- **Push(X)** – insert X as the top element of the stack
- **Pop()** – remove the top element of stack and return it.
- **Top()** – return the top element without removing it from the stack.
- **Peek** - This returns the top data value of the stack without removing it.
- **isFull:** Used to check whether the stack is full.
- **IsEmpty** - Checks if the stack is empty.

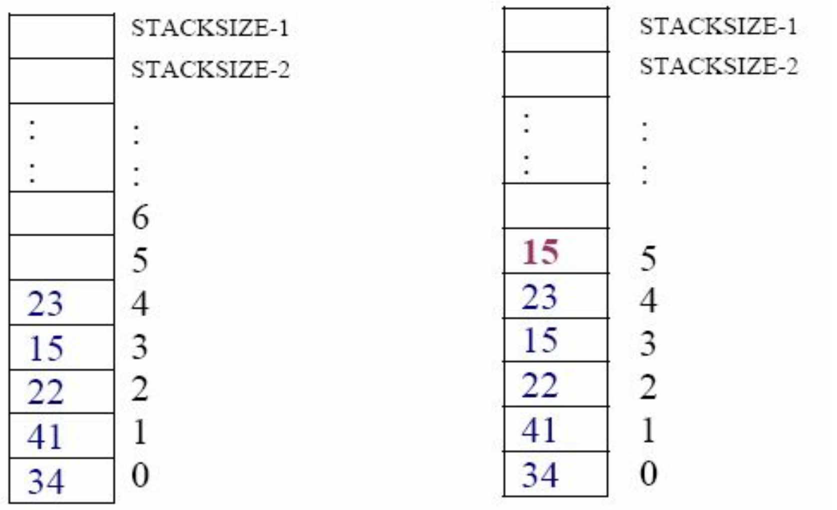
Stack Operations



Push Operation

- **Algorithm:** PUSH(STACK, TOP, STACKSIZE, ITEM)

```
1. [STACK already filled?]
      If TOP=STACKSIZE-1, then: Print: OVERFLOW / Stack Full, and Return.
2. Set TOP:=TOP+1. [Increase TOP by 1.]
3. Set STACK[TOP]=ITEM. [Insert ITEM in new TOP position.]
4. RETURN.
```



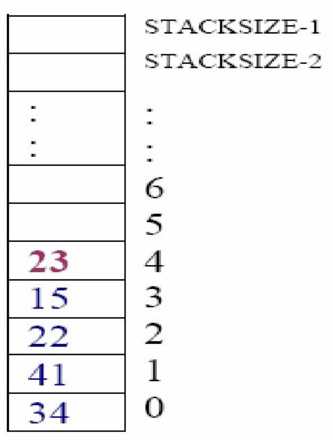
Before PUSH
(top=4, count=5)

After PUSH
(top=5, count= 6)

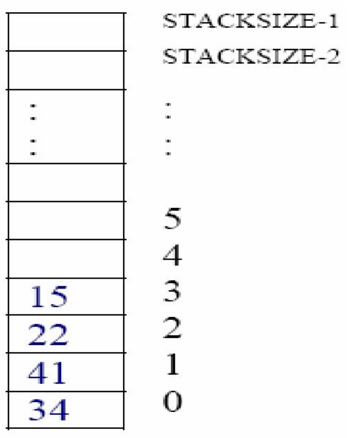
Pop Operation

- **Algorithm:** POP(STACK, TOP, ITEM)

```
1. [STACK has an item to be removed? Check for empty stack]
       If TOP=-1, then: Print: UNDERFLOW/ Stack is empty, and Return.
2. Set ITEM=STACK[TOP]. [Assign TOP element to ITEM.]
3. Set TOP=TOP-1. [Decrease TOP by 1.]
4. Return.
```



Before POP
(top=4, count=5)



After POP
(top=3 count=4)

Applications of Stack

- Stacks are used in conversion of infix to postfix expression.
- Stacks are also used in evaluation of postfix expression.
- Stacks are used to implement recursive procedures.
- Reverse String

Stacks Implementation

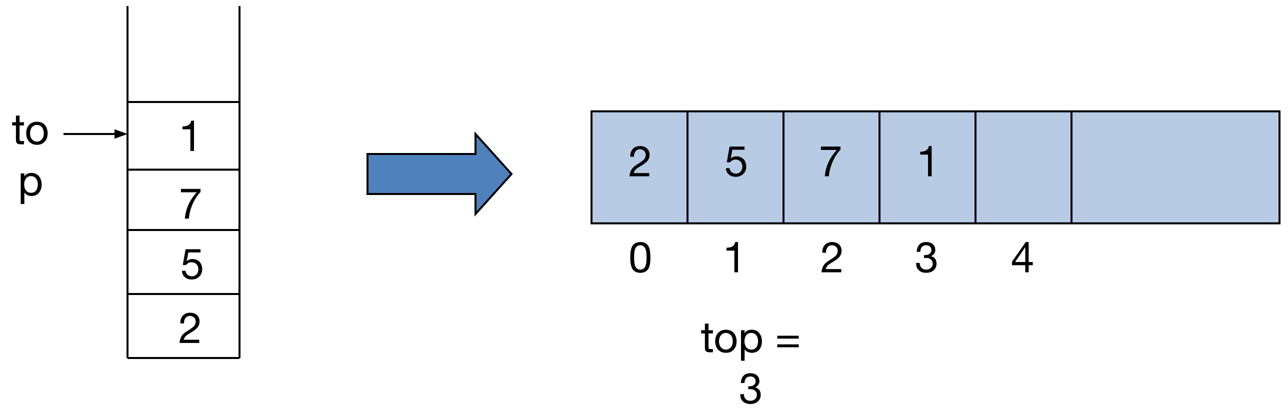
- **Array-based implementation**:

  It can uses the simple array to the store the elements of the stack. The pointer is used to the keep of the top of the stack.

- **Linked List based implementation**:

  Each element in the stack is the node in a linked list. The top of the stack is simply the head of the linked list.

Stack using an Array



Stack Operations with Array

```
void push(int x)
{
    arr[++top] = x;
}
int pop()
{
    return arr[top--];
}
```

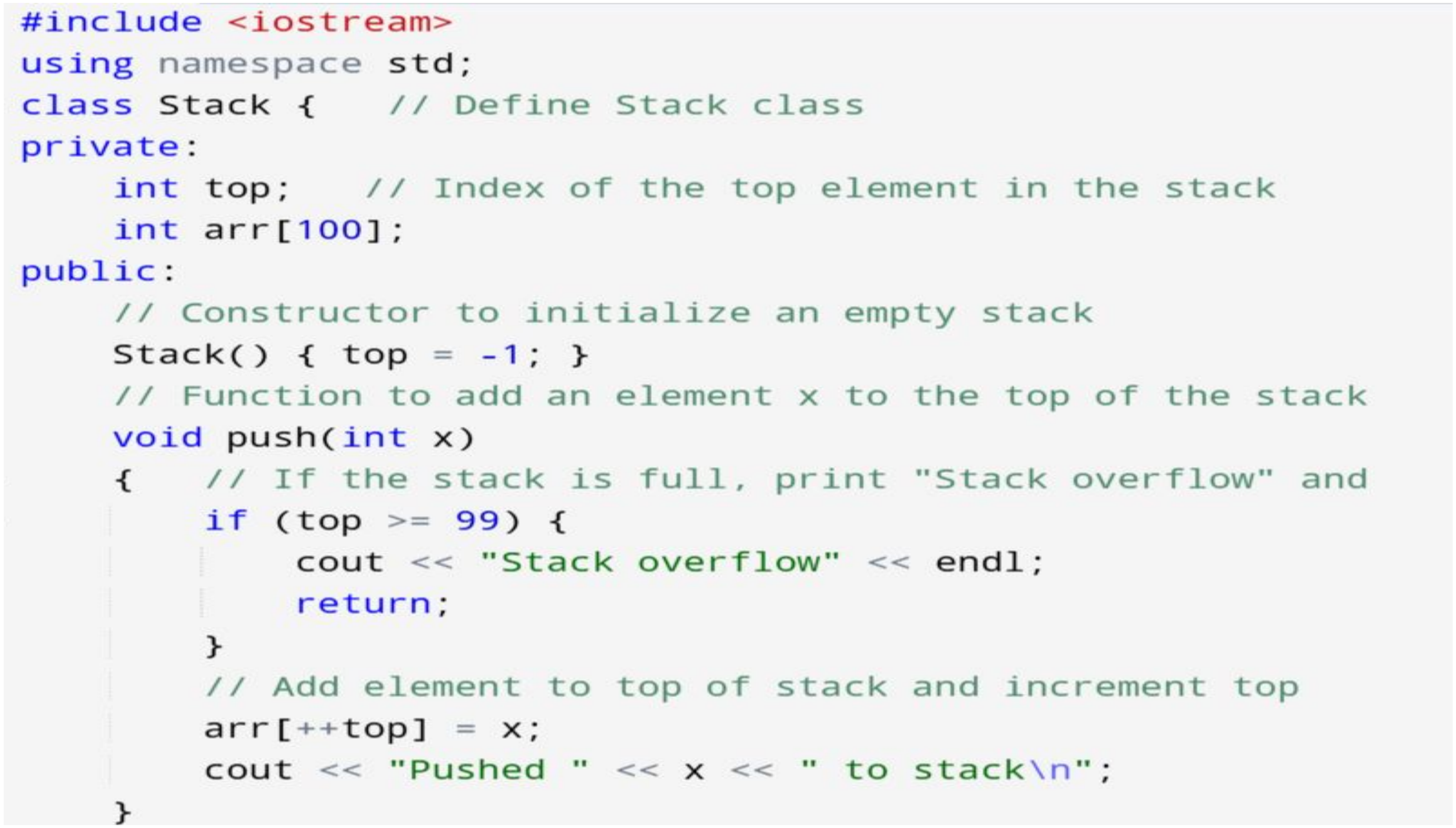
Stack Operations with Array

```
int top()
{
   return arr[top];
}

int IsEmpty()
{
   return ( top == -1 );
}

int IsFull()
{
   return ( top == size-1);
}
```

Peek- operation

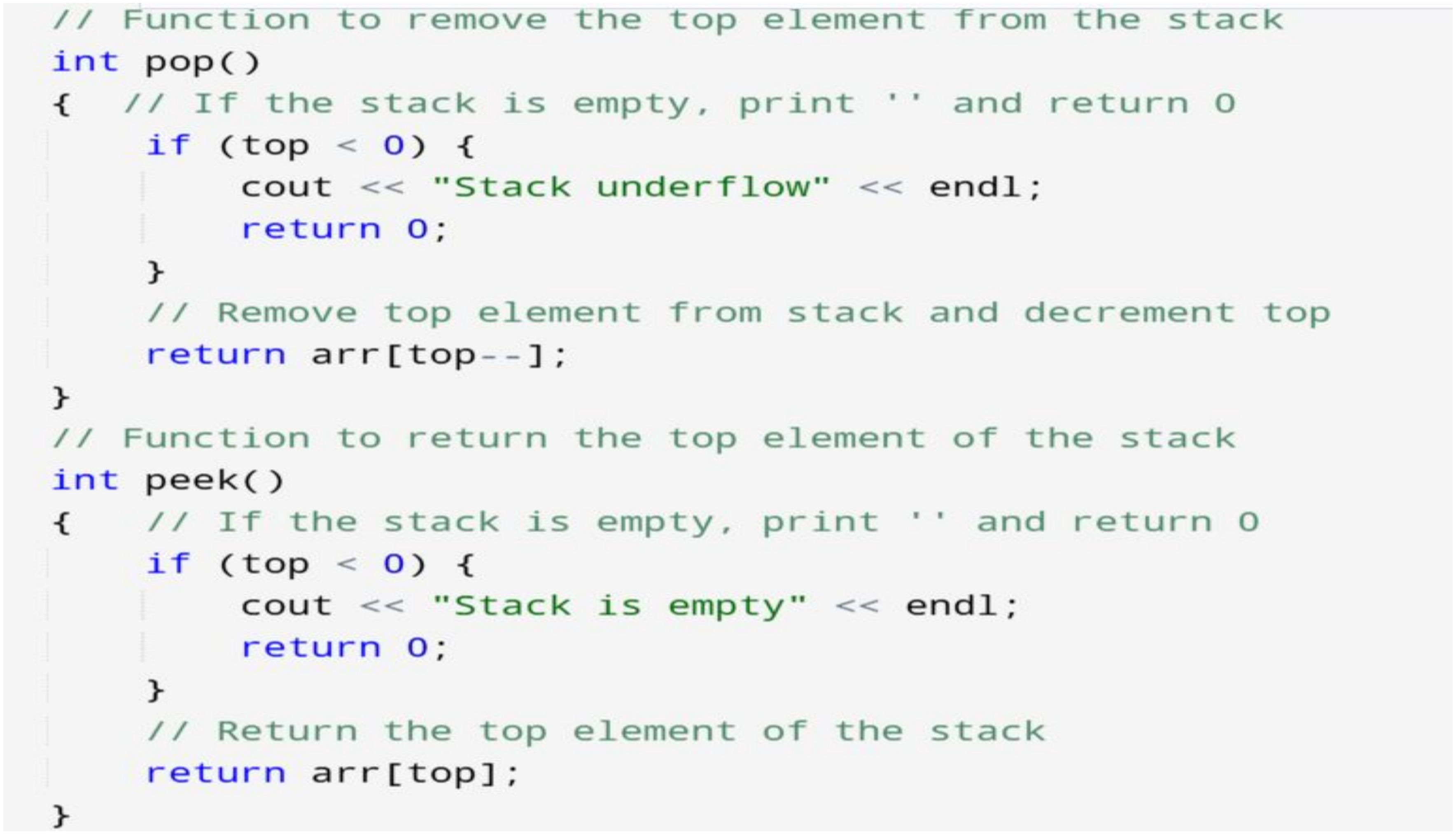
```
int peek()
  {if (top < 0) {
        cout << "Stack is empty" << endl;
        return 0;
     }
     // Return the top element of the stack
     return arr[top];
  }
```

Code



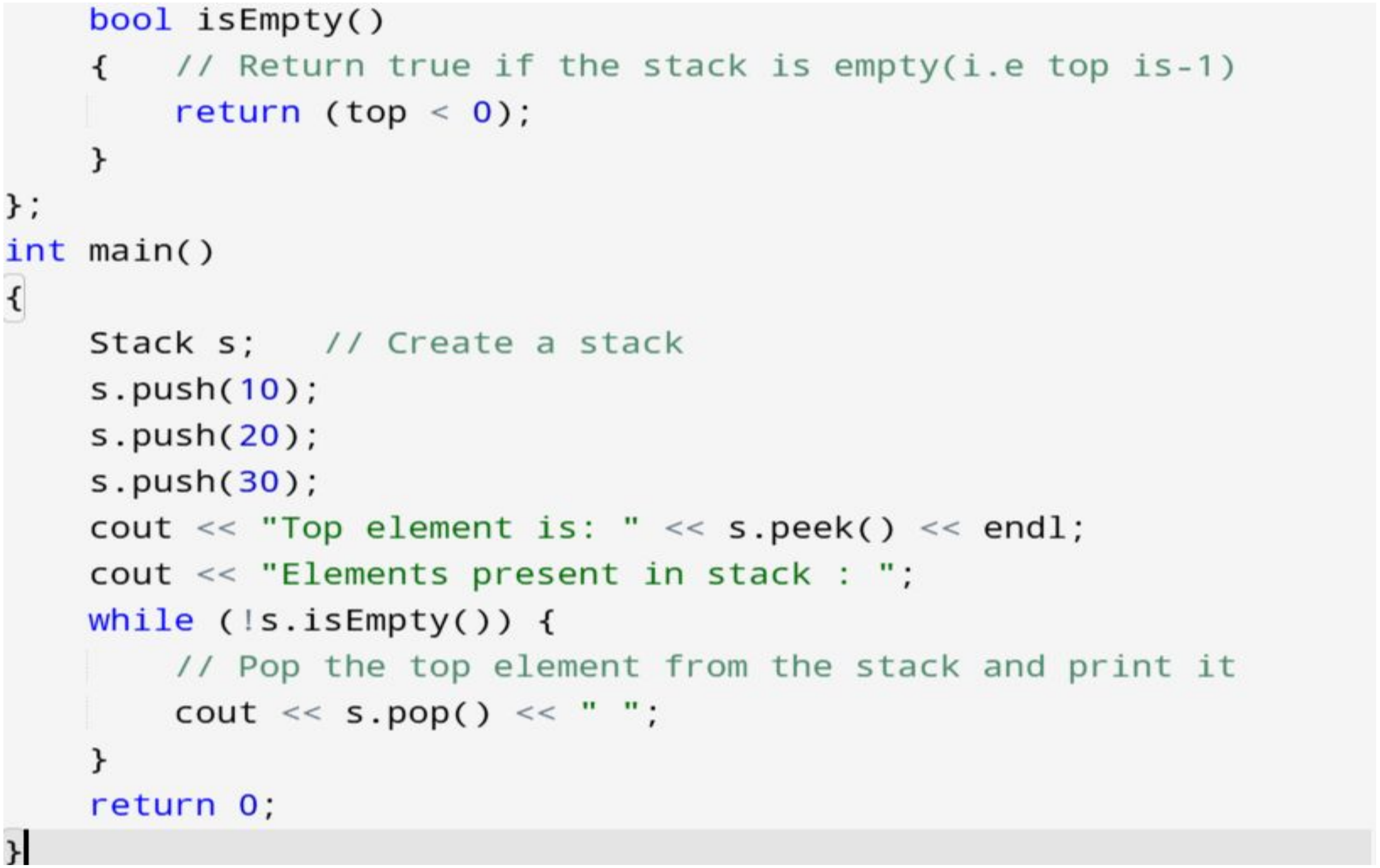
```cpp
#include <iostream>
using namespace std;
class Stack {   // Define Stack class
private:
    int top;   // Index of the top element in the stack
    int arr[100];
public:
    // Constructor to initialize an empty stack
    Stack() { top = -1; }
    // Function to add an element x to the top of the stack
    void push(int x)
    {   // If the stack is full, print "Stack overflow" and
        if (top >= 99) {
            cout << "Stack overflow" << endl;
            return;
        }
        // Add element to top of stack and increment top
        arr[++top] = x;
        cout << "Pushed " << x << " to stack\n";
    }
```

Code



```
// Function to remove the top element from the stack
int pop()
{  // If the stack is empty, print '' and return 0
    if (top < 0) {
        cout << "Stack underflow" << endl;
        return 0;
    }
    // Remove top element from stack and decrement top
    return arr[top--];
}
// Function to return the top element of the stack
int peek()
{   // If the stack is empty, print '' and return 0
    if (top < 0) {
        cout << "Stack is empty" << endl;
        return 0;
    }
    // Return the top element of the stack
    return arr[top];
}
```

Code



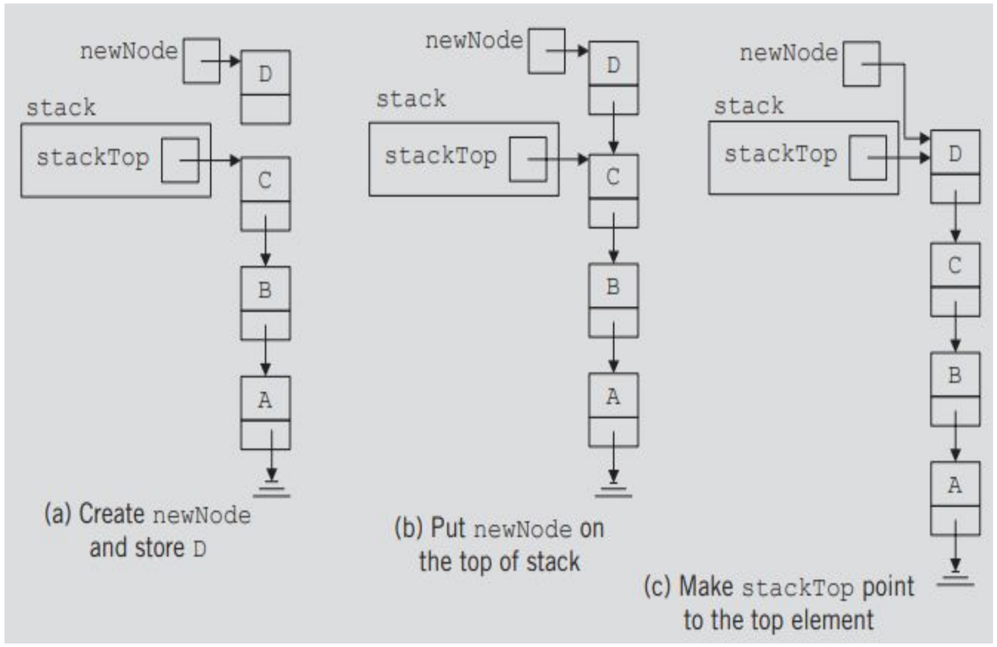
```cpp
    bool isEmpty()
    {   // Return true if the stack is empty(i.e top is-1)
        return (top < 0);
    }
};
int main()
{
    Stack s;    // Create a stack
    s.push(10);
    s.push(20);
    s.push(30);
    cout << "Top element is: " << s.peek() << endl;
    cout << "Elements present in stack : ";
    while (!s.isEmpty()) {
        // Pop the top element from the stack and print it
        cout << s.pop() << " ";
    }
    return 0;
}
```

Stack using STL

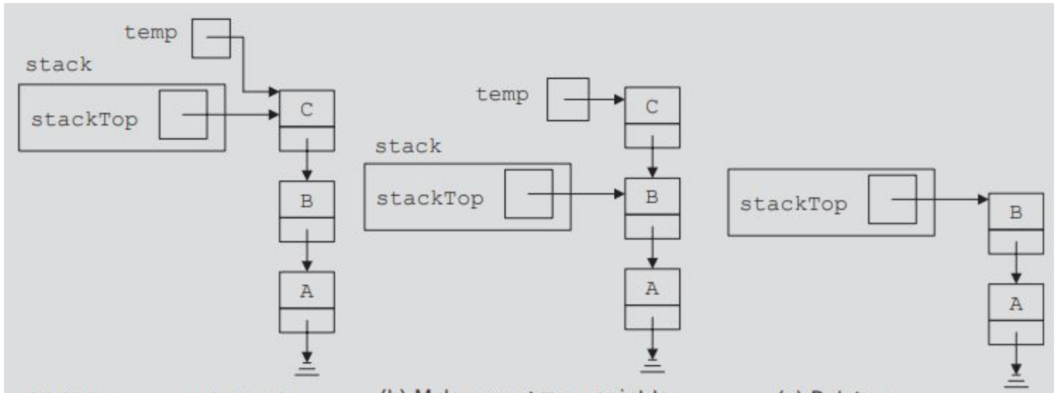
```
#include <iostream>
#include <stack>
using namespace std;

int main() {
  stack < int > s;
  for (int i = 0; i < 5; i++) {
    s.push(i + 1);
  }
  cout << "The size of stack is " << s.size() << '\n';
  cout << "The topmost element of the stack is " << s.top() << '\n';
  s.pop();
  s.pop();
  cout << "The elements of the stack after pop operation ";
  while (!s.empty()) {
    cout << s.top() << ' ';
    s.pop();
  }
  return 0;
}
```

Stack Implementation - Linked-List

```
void push(int new_data) {
      // Create a new node with given data
      Node* new_node = new Node(new_data);
      if (!new_node) {
         cout << "\nStack Overflow";
      }
      // Link the new node to the current top node
      new_node->next = head;

      // Update  top to  new node
      head = new_node;
   }
```


newNode
D
stack
stackTop
C
B
A
(a) Create newNode and store D
newNode
D
stack
stackTop
C
B
A
(b) Put newNode on the top of stack
newNode
stack
stackTop
D
C
B
A
(c) Make stackTop point to the top element

```
void pop() {
 if (this->isEmpty()) {
        cout << "\nStack Underflow" << endl;
     }
     else {
        // Assign the current top to a temporary  variable
        Node* temp = head;

        // Update the top to the next node
        head = head->next;

        // Deallocate the memory of the old top node
        delete temp;
     }
  }
```

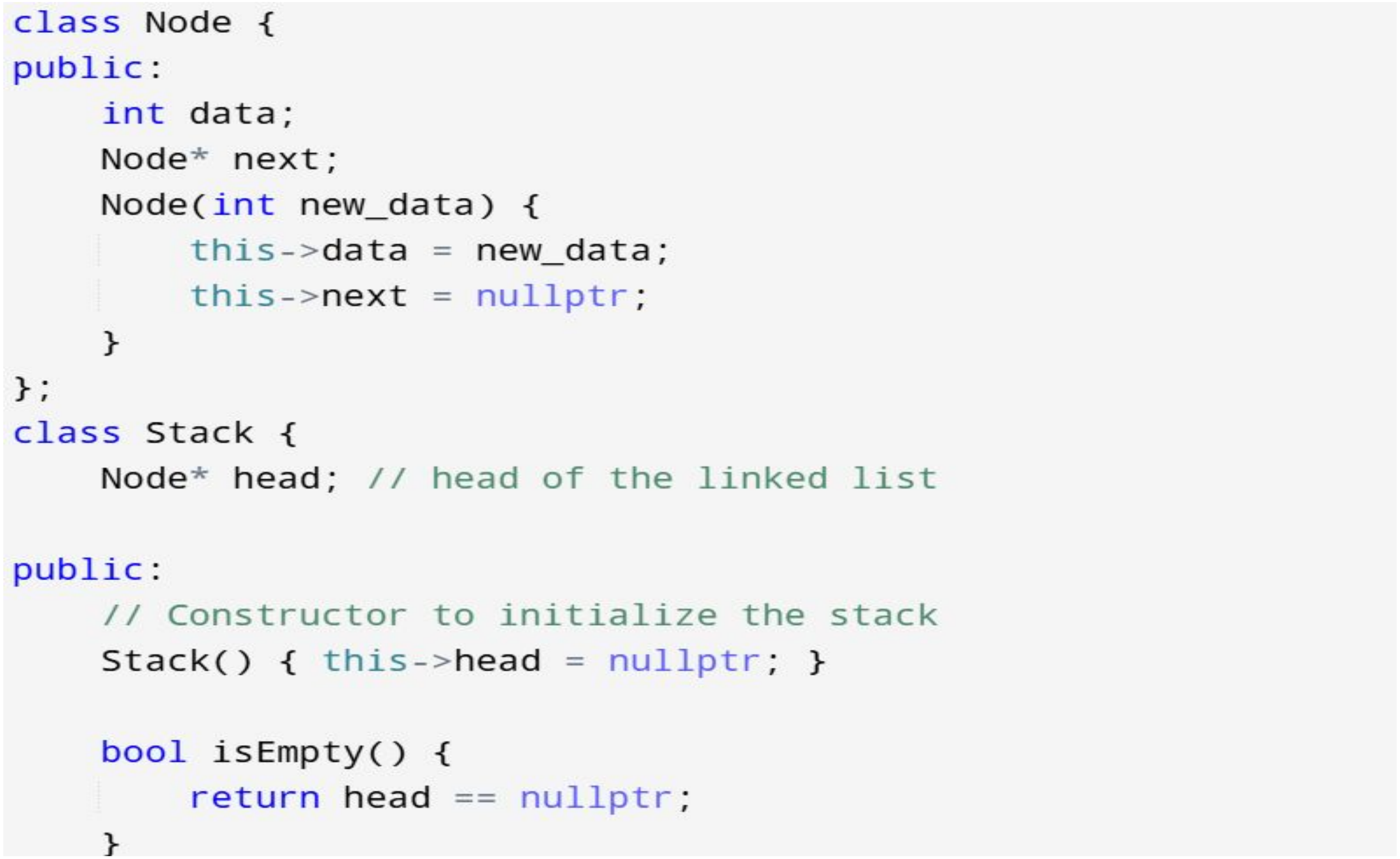


(a) Make temp point to the top element (b) Make stackTop point to the next element (c) Delete temp

Stack Implementation - Linked-List

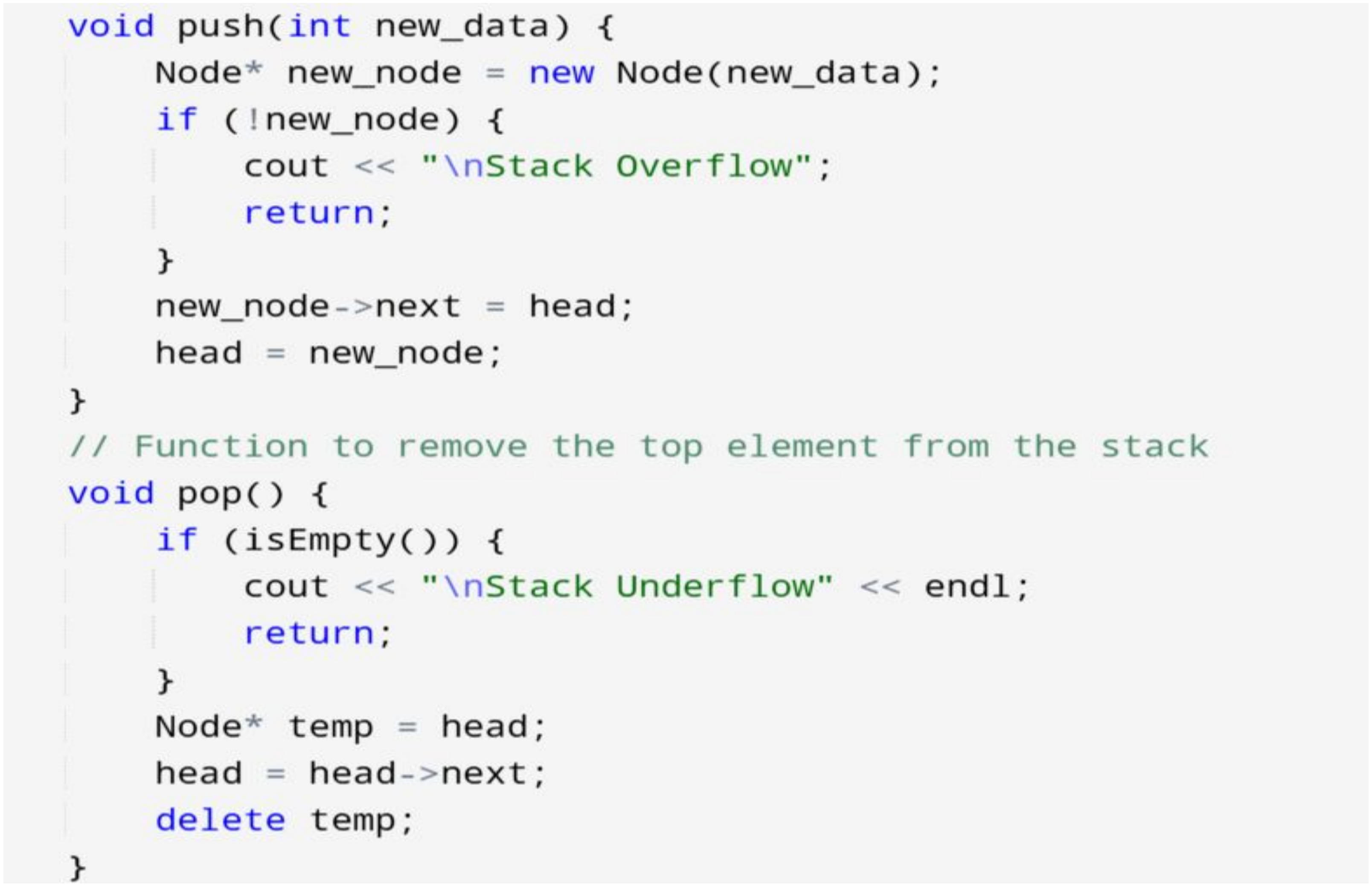
```
int peek() {
       // If stack is not empty, return the top element
       if (!isEmpty())
           return head->data;
       else {
             cout << "\nStack is empty";
               return INT_MIN;
            }
}
 bool isEmpty() {
       // If head is nullptr, the stack is empty
       return head == nullptr;
   }
```

Code



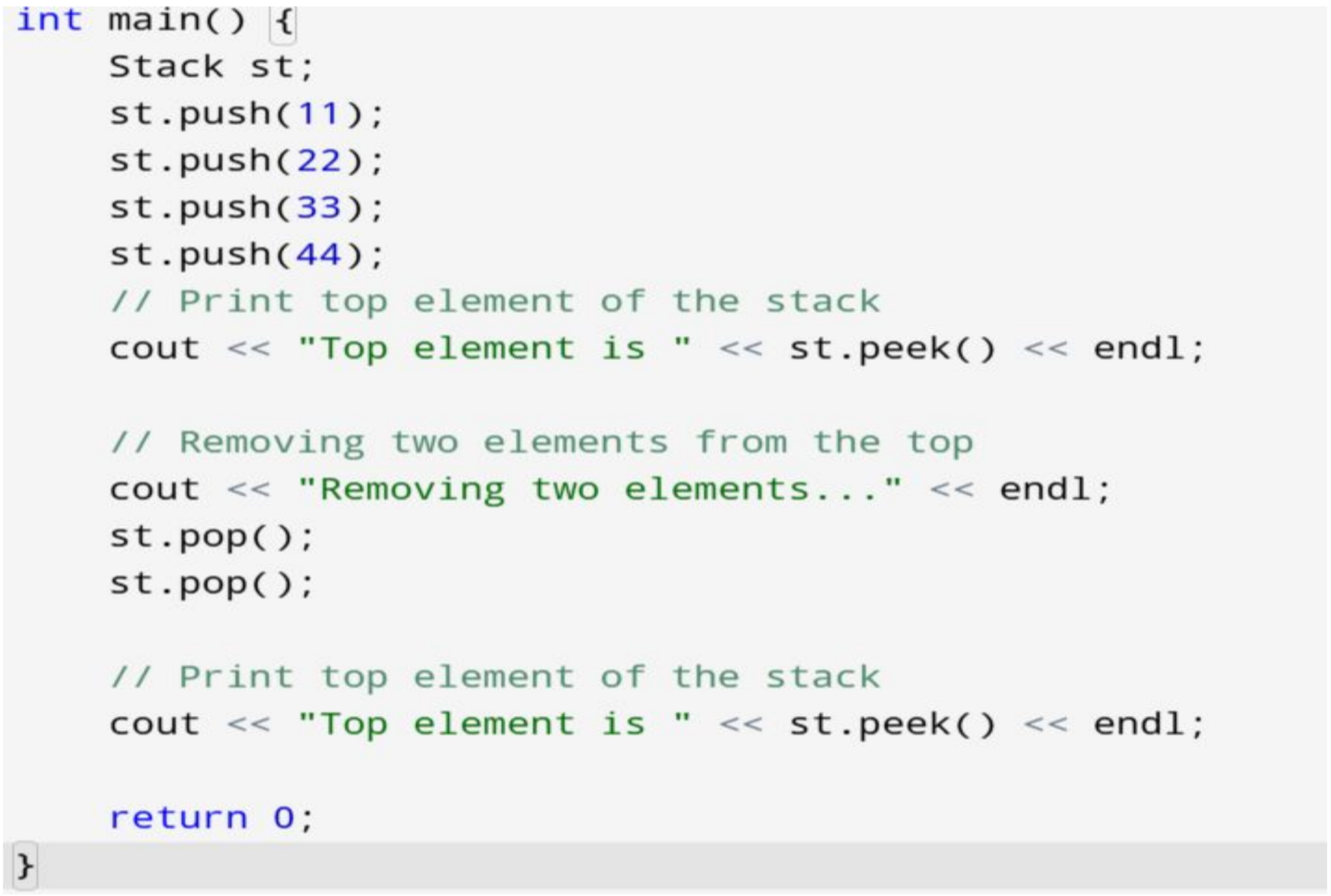
```
class Node {
public:
    int data;
    Node* next;
    Node(int new_data) {
        this->data = new_data;
        this->next = nullptr;
    }
};
class Stack {
    Node* head; // head of the linked list

public:
    // Constructor to initialize the stack
    Stack() { this->head = nullptr; }

    bool isEmpty() {
        return head == nullptr;
    }
```

Code



```
void push(int new_data) {
    Node* new_node = new Node(new_data);
    if (!new_node) {
        cout << "\nStack Overflow";
        return;
    }
    new_node->next = head;
    head = new_node;
}
// Function to remove the top element from the stack
void pop() {
    if (isEmpty()) {
        cout << "\nStack Underflow" << endl;
        return;
    }
    Node* temp = head;
    head = head->next;
    delete temp;
}
```

Code



```cpp
int main() {
    Stack st;
    st.push(11);
    st.push(22);
    st.push(33);
    st.push(44);
    // Print top element of the stack
    cout << "Top element is " << st.peek() << endl;

    // Removing two elements from the top
    cout << "Removing two elements..." << endl;
    st.pop();
    st.pop();

    // Print top element of the stack
    cout << "Top element is " << st.peek() << endl;

    return 0;
}
```

Using a Stack to Process Algebraic Expressions

- Algebraic expressions composed of
  - Operands (variables, constants)
  - Operators (+, -, /, *, ^)
- Operators can be unary or binary
- Different precedence notations
  - Infix a + b
  - Prefix + a b
  - Postfix a b +

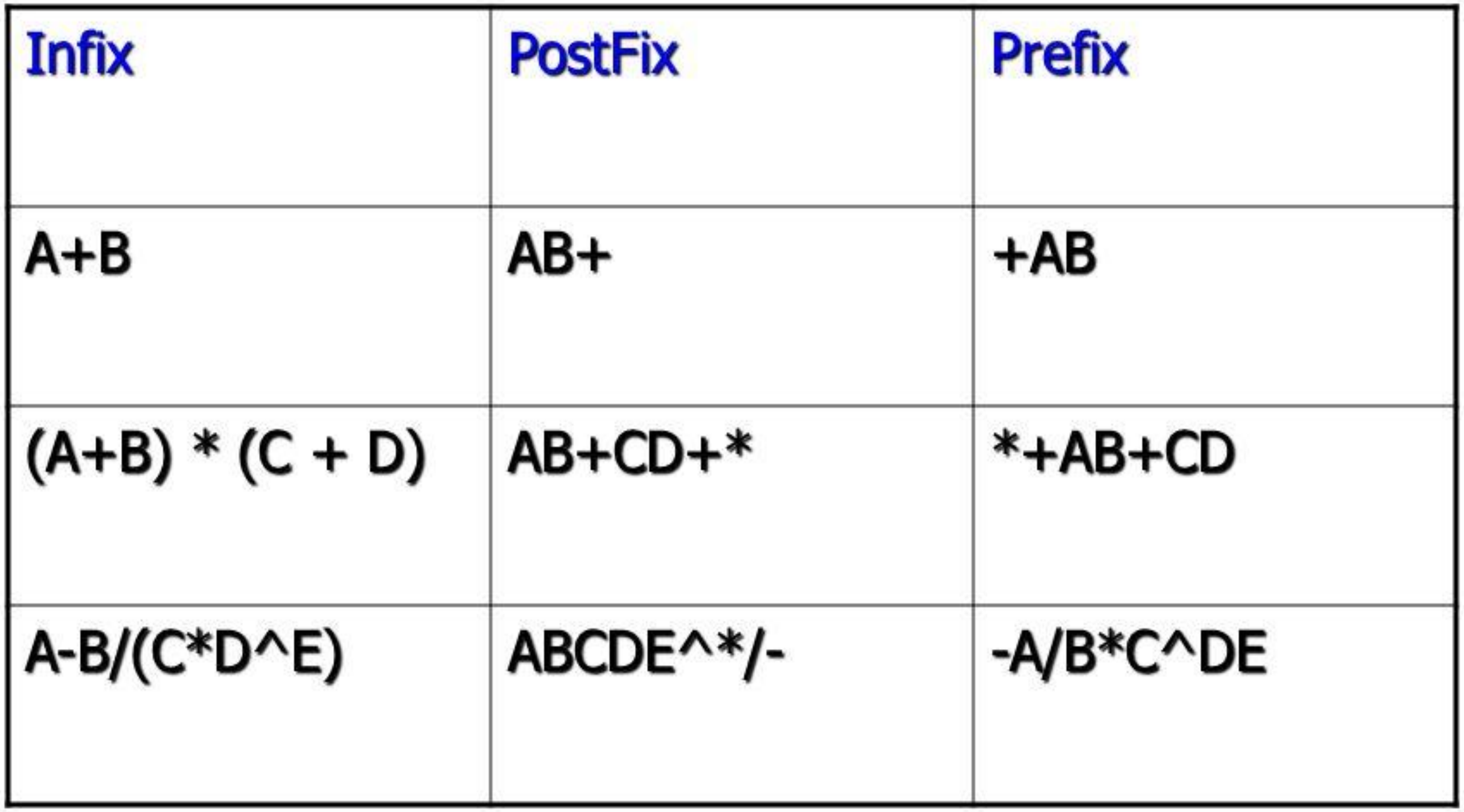
Precedence of Operators

- The order of precedence is (highest to lowest)
  - Exponentiation ↑
  - Multiplication/division *, /
  - Addition/subtraction +, -

Precedence of Operators

- For operators of same precedence, the **left-to-right rule** applies:
  $A+B+C$ means $(A+B)+C$.
- For exponentiation, the **right-to-left** rule applies
  $A \uparrow B \uparrow C$ means $A \uparrow (B \uparrow C)$

Example



Infix	PostFix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Infix to Postfix

- Conversion to postfix

  A + (B * C)   infix form

Infix to Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication

Infix to Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication
A (B C *) +	convert addition

Infix to Postfix

- Conversion to postfix

A + (B * C)	infix form
A + (B C *)	convert multiplication
A (B C *) +	convert addition
A B C * +	postfix form

Infix to Postfix

- The postfix form an expression does not require parenthesis.
- Consider ‘4+3*5’ and ‘(4+3)*5’. The parenthesis are not needed in the first but they are necessary in the second.
- The postfix forms are:

4+3*5	435*+
(4+3)*5	43+5*

Infix to Postfix Algo using Stack

- Scan the infix expression **from left to right**.
- If the scanned character is an operand, put it in the postfix expression.
- Otherwise, do the following
  - If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.
  - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.
- If the scanned character is a '**(**', push it to the stack.
- If the scanned character is a '**)**', pop the stack and output it until a '**(**' is encountered, and discard both the parenthesis.
- Repeat steps until the infix expression is scanned.

Converting Infix to Postfix

- **Example:** A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC * +	
	ABC * +	

$a - (b + c * d)/e$ to

ch	stack (bottom to top)	postfixExp
a		a
−	−	a
(	− (	a
b	− (	ab
+	− (+	ab
c	− (+	abc
*	− (+ *	abc
d	− (+ *	abcd
)	− (+	abcd*
	− (	abcd*+
	−	abcd*+
/	− /	abcd*+
e	− /	abcd*+e
		abcd*+e/−

Practice task

- ***Input:*** *((A + B) – C * (D / E)) + F*
-

  Output: *AB+CDE/*-F+*

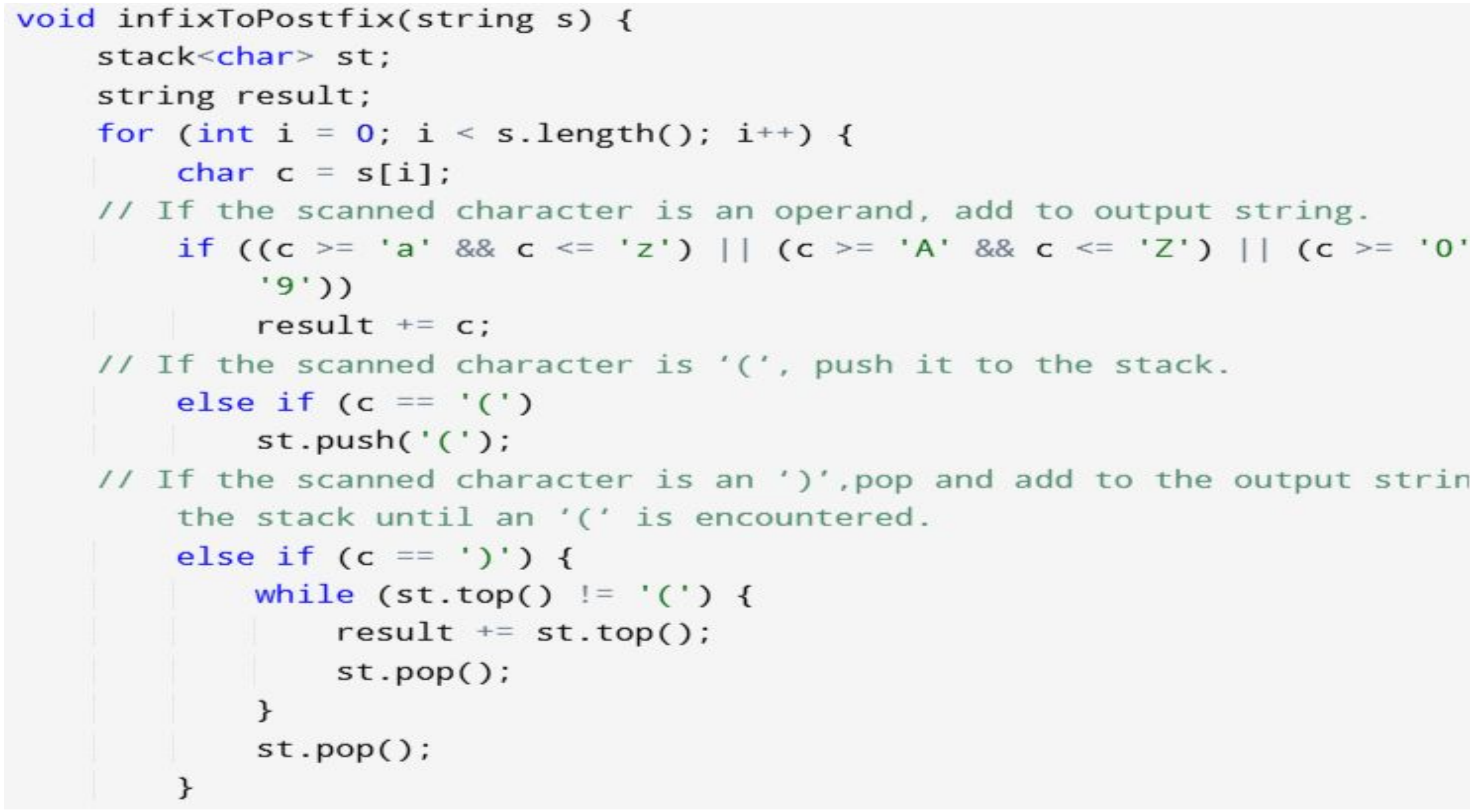
- ***Input:*** A + ((B + C) * (E - F) - G) / (H - I);

  Output: A B C + E F - * G - H I - / +

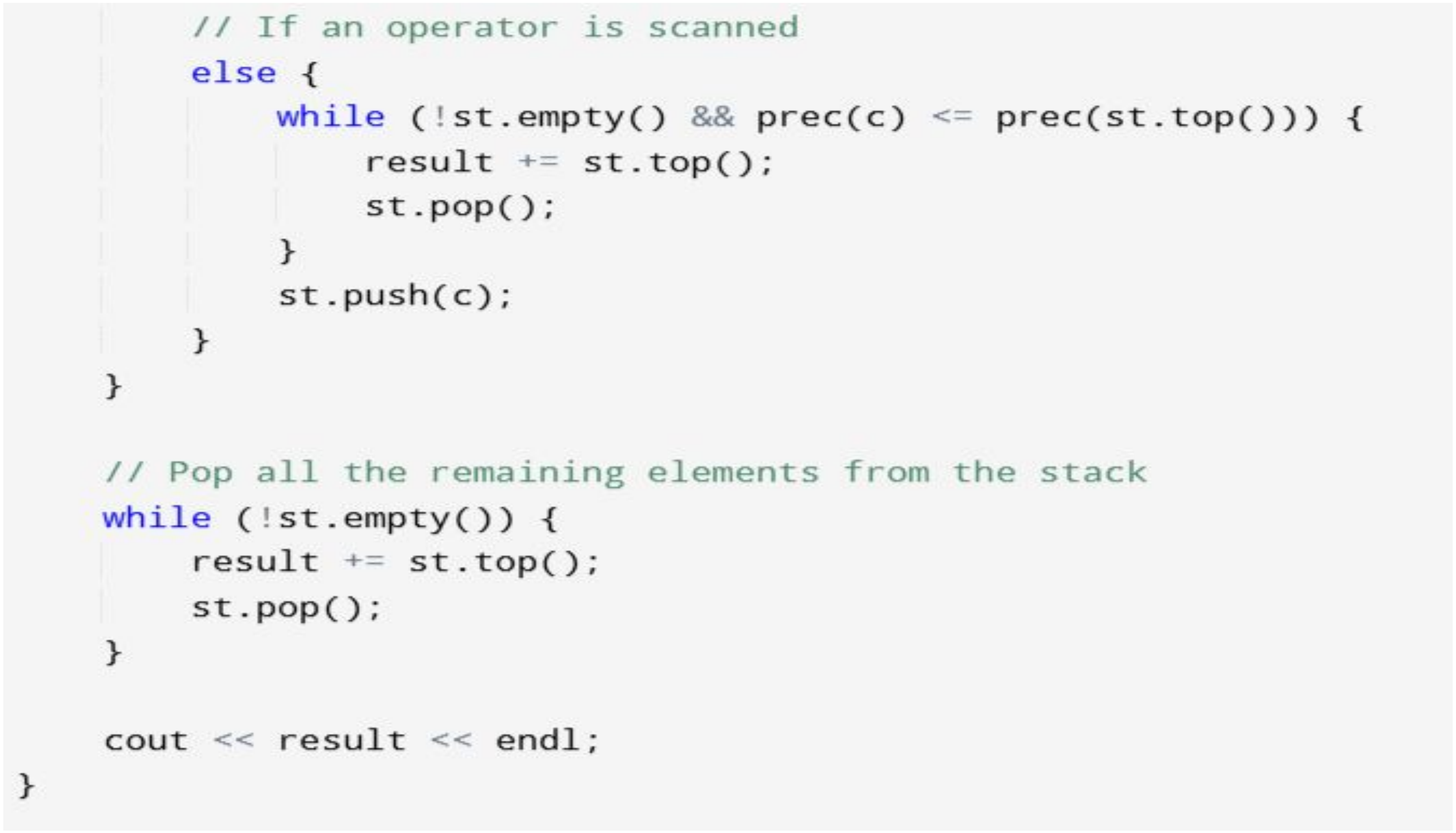
Code

```
int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}
```

Infix to Postfix



```
void infixToPostfix(string s) {
    stack<char> st;
    string result;
    for (int i = 0; i < s.length(); i++) {
        char c = s[i];
    // If the scanned character is an operand, add to output string.
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0'
            '9'))
            result += c;
    // If the scanned character is ‘(‘, push it to the stack.
        else if (c == '(')
            st.push('(');
    // If the scanned character is an ‘)’,pop and add to the output strin
        the stack until an ‘(‘ is encountered.
        else if (c == ')') {
            while (st.top() != '(') {
                result += st.top();
                st.pop();
            }
            st.pop();
        }
```



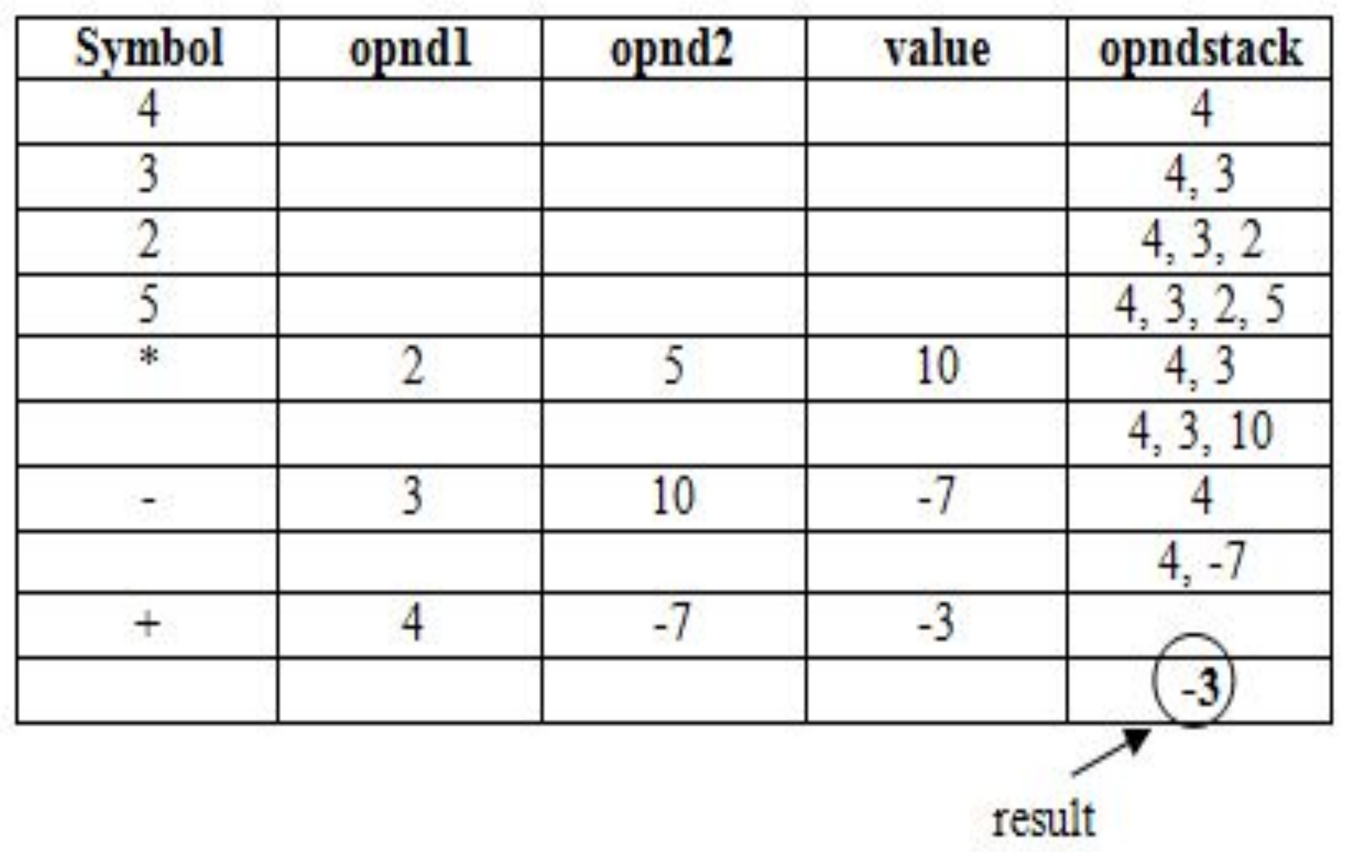
```
        // If an operator is scanned
        else {
            while (!st.empty() && prec(c) <= prec(st.top())) {
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }

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

    cout << result << endl;
}
```

Evaluating Postfix

- Postfix is also called '**Reverse Polish Notation**'
- **Algo:**
  - Create a stack to store operands (or values).
  - Scan the given expression from left to right and do the following for every scanned element.
    - If the element is a number, push it into the stack.
    - If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
  - When the expression is ended, the number in the stack is the final answer.

Example

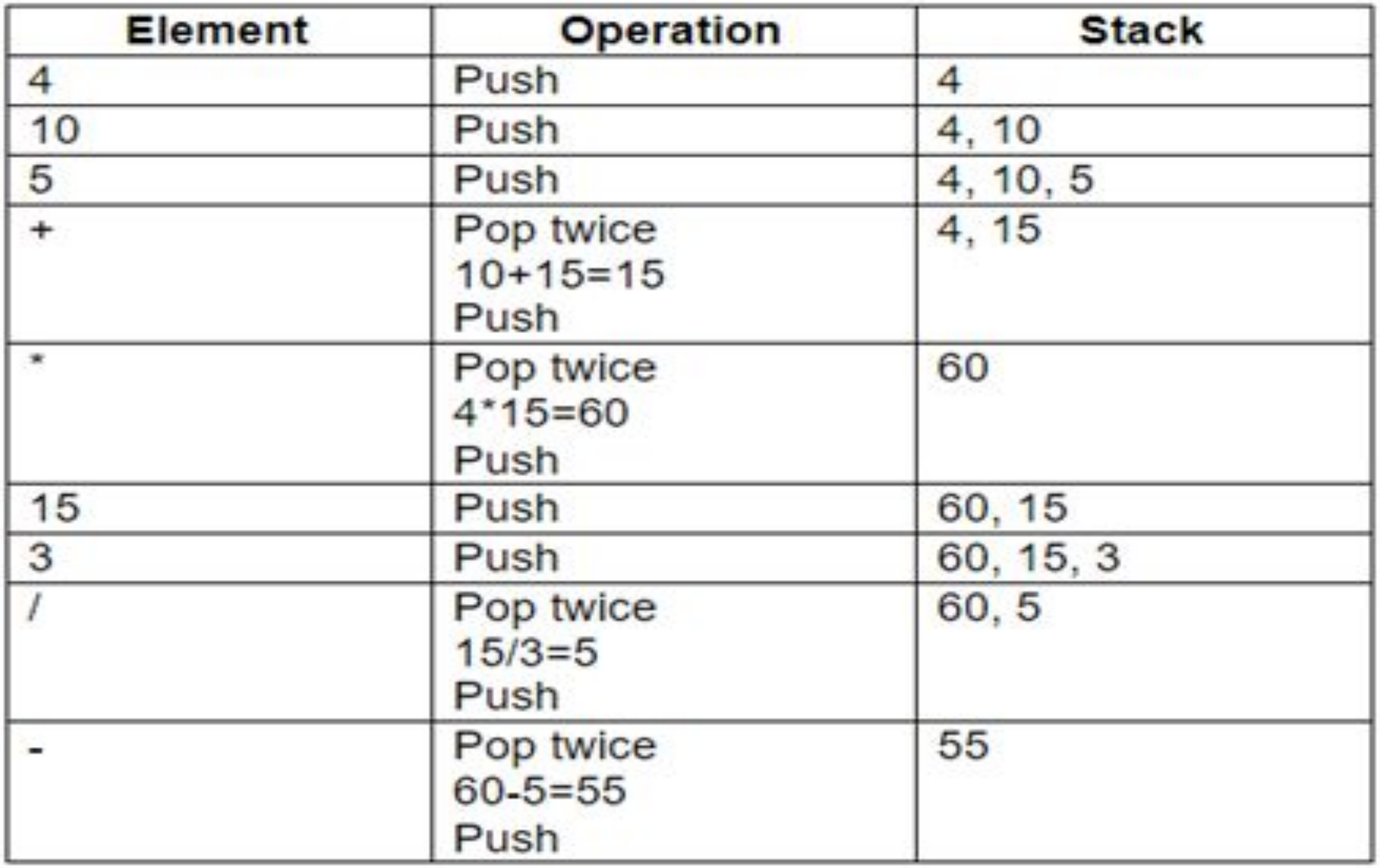
4325*-+ = 4+3-2*5



Symbol	opnd1	opnd2	value	opndstack
4				4
3				4, 3
2				4, 3, 2
5				4, 3, 2, 5
*	2	5	10	4, 3
				4, 3, 10
-	3	10	-7	4
				4, -7
+	4	-7	-3	
				-3

result

Example



Element	Operation	Stack
4	Push	4
10	Push	4, 10
5	Push	4, 10, 5
+	Pop twice 10+15=15 Push	4, 15
*	Pop twice 4*15=60 Push	60
15	Push	60, 15
3	Push	60, 15, 3
/	Pop twice 15/3=5 Push	60, 5
-	Pop twice 60-5=55 Push	55

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6			6	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6			**6**	
2			**6,2**	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6			**6**	
2			**6,2**	
3			**6,2,3**	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6			6	
2			6,2	
3			6,2,3	
+	2	3	5	6,5

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value		stack
6				6		
2				6,2		
3				6,2,3		
+		2	3	5	6,5	
-	6	5	1	1		

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		**op1**	**op2**	**value**		**stack**
6				**6**		
2				**6,2**		
3				**6,2,3**		
+		**2**	**3**	**5**	**6,5**	
-	**6**	**5**	**1**	**1**		
3	**6**	**5**	**1**	**1,3**		

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6				6	
2				6,2	
3				6,2,3	
+		2	3	5	6,5
-	6	5	1	1	
3	6	5	1	1,3	
8	6	5	1	1,3,8	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		**op1**	**op2**	**value**		**stack**
6				**6**		
2				**6,2**		
3				**6,2,3**		
+		**2**	**3**	**5**	**6,5**	
-	**6**	**5**	**1**	**1**		
3	**6**	**5**	**1**	**1,3**		
8	**6**	**5**	**1**	**1,3,8**		
2	**6**	**5**	**1**	**1,3,8,2**		

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		**op1**	**op2**	**value**	**stack**
6				**6**	
2				**6,2**	
3				**6,2,3**	
+		**2**	**3**	**5**	**6,5**
-	**6**	**5**	**1**	**1**	
3	**6**	**5**	**1**	**1,3**	
8	**6**	**5**	**1**	**1,3,8**	
2	**6**	**5**	**1**	**1,3,8,2**	
/	**8**	**2**	**4**	**1,3,4**	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value		stack
6				6		
2				6,2		
3				6,2,3		
+		2	3	5	6,5	
-	6	5	1	1		
3	6	5	1	1,3		
8	6	5	1	1,3,8		
2	6	5	1	1,3,8,2		
/	8	2	4	1,3,4		
+		3	4	7	1,7	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6				6	
2				6,2	
3				6,2,3	
+		2	3	5	6,5
-	6	5	1	1	
3	6	5	1	1,3	
8	6	5	1	1,3,8	
2	6	5	1	1,3,8,2	
/	8	2	4	1,3,4	
+		3	4	7	1,7
*	1	7	7	7	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6				6	
2				6,2	
3				6,2,3	
+		2	3	5	6,5
-	6	5	1	1	
3	6	5	1	1,3	
8	6	5	1	1,3,8	
2	6	5	1	1,3,8,2	
/	8	2	4	1,3,4	
+		3	4	7	1,7
*	1	7	7	7	
2	1	7	7	7,2	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6				6	
2				6,2	
3				6,2,3	
+		2	3	5	6,5
-	6	5	1	1	
3	6	5	1	1,3	
8	6	5	1	1,3,8	
2	6	5	1	1,3,8,2	
/	8	2	4	1,3,4	
+		3	4	7	1,7
*	1	7	7	7	
2	1	7	7	7,2	
↑	7	2	49	49	
3	7	2	49	49,3	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6				6	
2				6,2	
3				6,2,3	
+		2	3	5	6,5
-	6	5	1	1	
3	6	5	1	1,3	
8	6	5	1	1,3,8	
2	6	5	1	1,3,8,2	
/	8	2	4	1,3,4	
+		3	4	7	1,7
*	1	7	7	7	
2	1	7	7	7,2	
↑	7	2	49	49	
3	7	2	49	49,3	
+		49	3	52	52

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6				6	
2				6,2	
3				6,2,3	
+		2	3	5	6,5
-	6	5	1	1	
3	6	5	1	1,3	
8	6	5	1	1,3,8	
2	6	5	1	1,3,8,2	
/	8	2	4	1,3,4	
+		3	4	7	1,7
*	1	7	7	7	
2	1	7	7	7,2	
↑	7	2	49	49	
3	7	2	49	49,3	
+		49	3	52	52

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		**op1**	**op2**	**value**	**stack**
6				**6**	
2				**6,2**	
3				**6,2,3**	
+		**2**	**3**	**5**	**6,5**
-	**6**	**5**	**1**	**1**	
3	**6**	**5**	**1**	**1,3**	
8	**6**	**5**	**1**	**1,3,8**	
2	**6**	**5**	**1**	**1,3,8,2**	
/	**8**	**2**	**4**	**1,3,4**	
+		**3**	**4**	**7**	**1,7**
*****	**1**	**7**	**7**	**7**	
2	**1**	**7**	**7**	**7,2**	
↑	**7**	**2**	**49**	**49**	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6				6	
2				6,2	
3				6,2,3	
+		2	3	5	6,5
-	6	5	1	1	
3	6	5	1	1,3	
8	6	5	1	1,3,8	
2	6	5	1	1,3,8,2	
/	8	2	4	1,3,4	
+		3	4	7	1,7
*	1	7	7	7	
2	1	7	7	7,2	
↑	7	2	49	49	
3	7	2	49	49,3	

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input		op1	op2	value	stack
6				6	
2				6,2	
3				6,2,3	
+		2	3	5	6,5
-	6	5	1	1	
3	6	5	1	1,3	
8	6	5	1	1,3,8	
2	6	5	1	1,3,8,2	
/	8	2	4	1,3,4	
+		3	4	7	1,7
*	1	7	7	7	
2	1	7	7	7,2	
↑	7	2	49	49	
3	7	2	49	49,3	
+		49	3	52	52

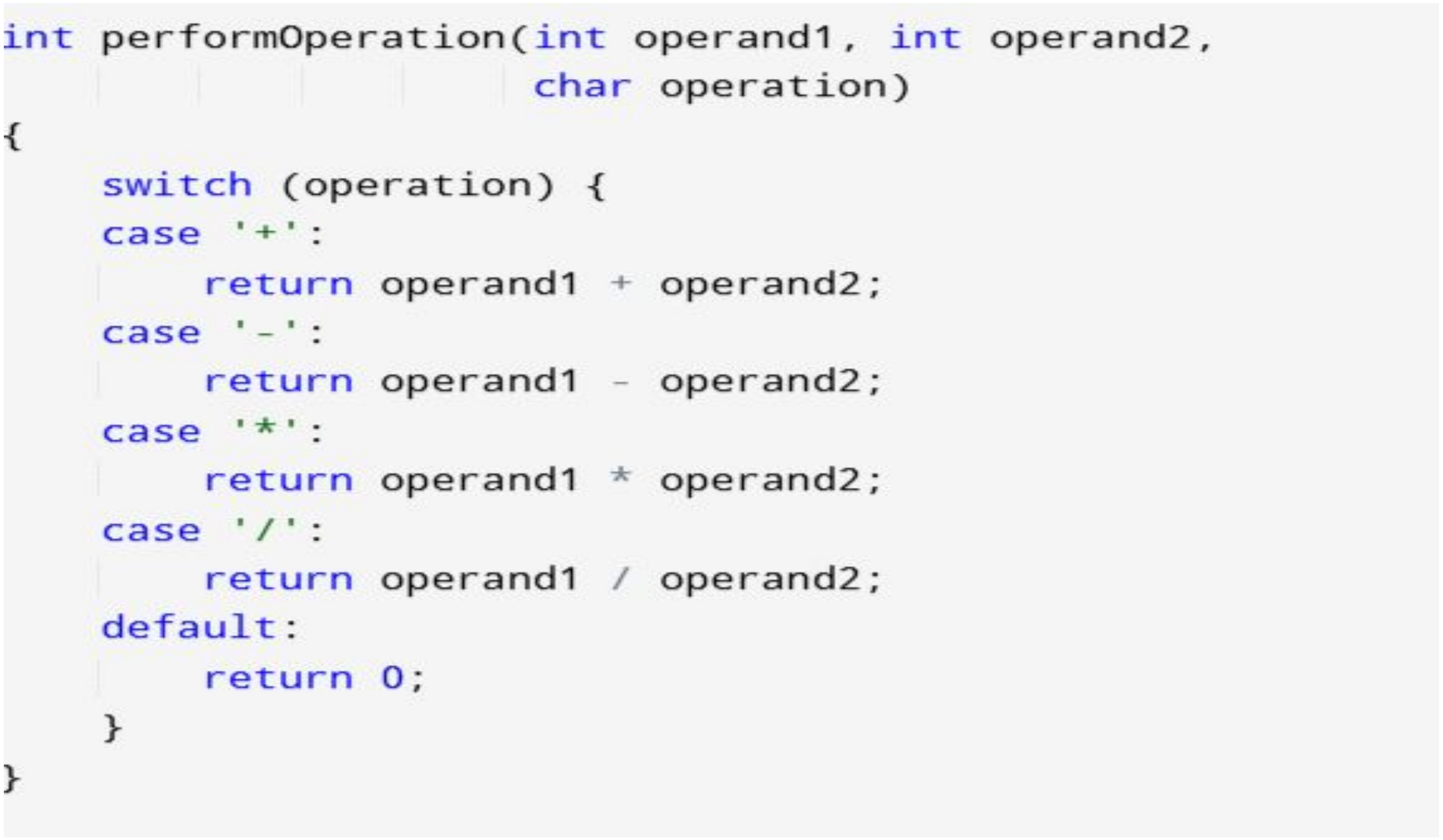
Practice Problem

- 231*+9-
- 73*4+
- *100 200 + 2 / 5 * 7 +*

Evaluating Postfix

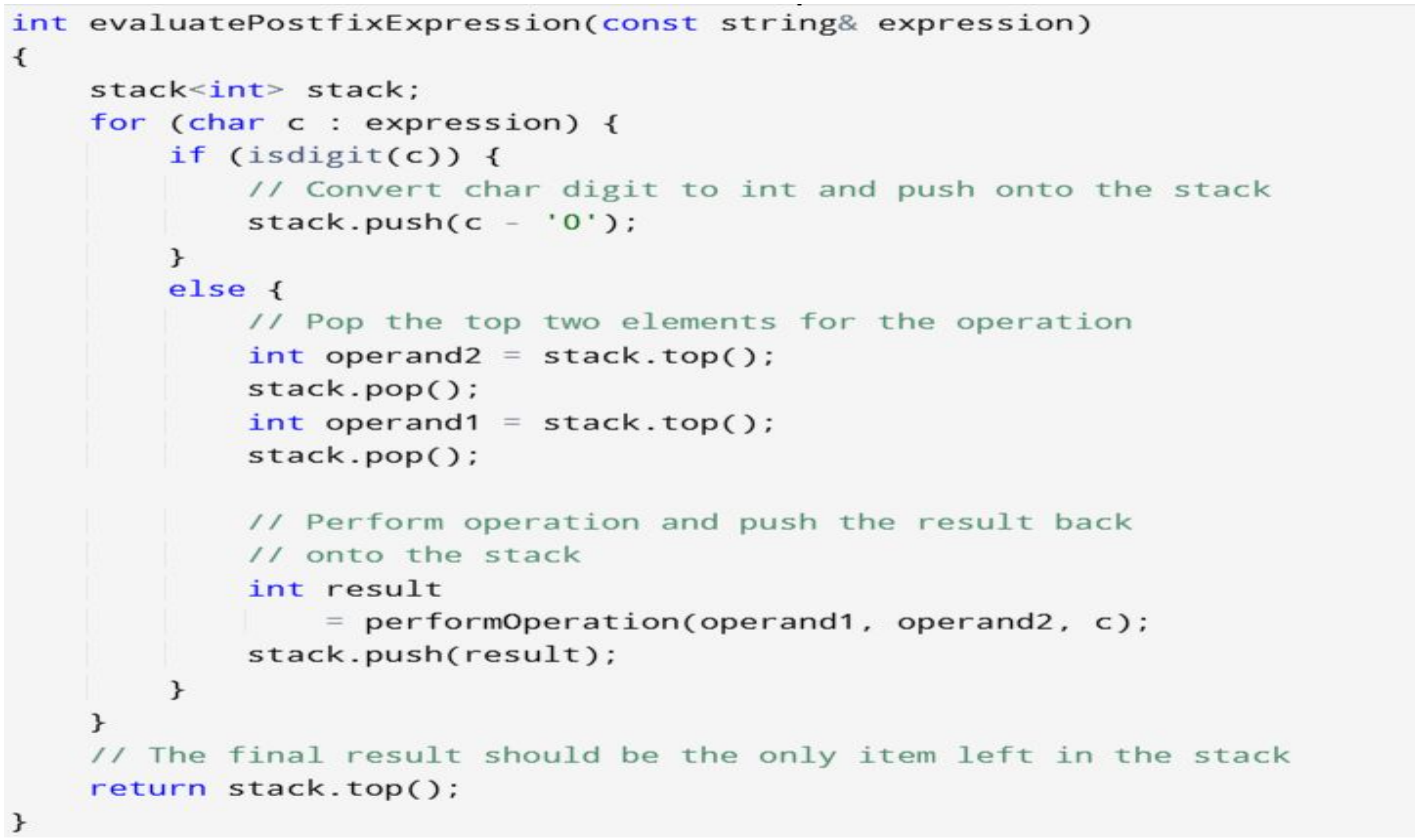
```
Stack s;
while( not end of input ) {
    e = get next element of input
    if( e is an operand )
        s.push( e );
    else {
        op2 = s.pop();
        op1 = s.pop();
        value = result of applying operator ‘e’ to op1 and op2;
        s.push( value );
    }
}
finalresult = s.pop();
```

Code



```
int performOperation(int operand1, int operand2,
                     char operation)
{
    switch (operation) {
    case '+':
        return operand1 + operand2;
    case '-':
        return operand1 - operand2;
    case '*':
        return operand1 * operand2;
    case '/':
        return operand1 / operand2;
    default:
        return 0;
    }
}
```

Code



```
int evaluatePostfixExpression(const string& expression)
{
    stack<int> stack;
    for (char c : expression) {
        if (isdigit(c)) {
            // Convert char digit to int and push onto the stack
            stack.push(c - '0');
        }
        else {
            // Pop the top two elements for the operation
            int operand2 = stack.top();
            stack.pop();
            int operand1 = stack.top();
            stack.pop();

            // Perform operation and push the result back
            // onto the stack
            int result
                = performOperation(operand1, operand2, c);
            stack.push(result);
        }
    }
    // The final result should be the only item left in the stack
    return stack.top();
}
```

Guess the Output?

```
stackType<int> stack;
int x, y;
x = 4;
y = 0;
stack.push(7);
stack.push(x);
stack.push(x + 5);
y = stack.top();
stack.pop();
stack.push(x + y);
stack.push(y - 2);
stack.push(3);
x = stack.top();
stack.pop();
cout << "x = " << x << endl;
cout << "y = " << y << endl;
while (!stack.isEmptyStack())
{
cout << stack.top() << endl;
stack.pop();
}
```

Practice Task

- Evaluate the following **postfix expressions**:
  a. 8 2 + 3 * 16 4 / -
  b. 12 25 5 1 / / * 8 7 + -
  c. 70 14 4 5 15 3 / * - - / 6 +
  d. 3 5 6 * + 13 - 18 2 / +
- Evaluate the following **infix expressions**:
  a. A + B - C;
  b. (A + B) * (C - D);
  c. A + B * (C + D) - E / F * G + H;

Reverse a string

```
#include <iostream>
#include <stack>
using namespace std;

string reverseString(string str) {
   stack<char> s;
for (char c : str) {      // Push all characters onto the stack
      s.push(c);
   }
 string reversed = "";   // Pop characters from the stack to reverse the string
   while (!s.empty()) {
      reversed += s.top();
      s.pop();
   }
   return reversed;
}
```

```
int main() {
    string input;
    cout << "Enter a string: ";
    cin >> input;

    string reversed = reverseString(input);
    cout << "Reversed string: " << reversed << endl;

    return 0;
}
```