STACK (Abstract Data Type) — Complete Notes in C++

1. What is a Stack (Abstract Data Type)?

A **Stack** is a **linear data structure** that follows the principle:

LIFO (Last In, First Out)

This means:

- The **last element inserted** is the **first one to be removed**.

Key Characteristics:

- Insertion and deletion happen at **one end only** → called **TOP**
- No random access like arrays
- Very useful in recursive problems and expression evaluation

Real-Life Analogies

1. **Stack of Plates** 🍽️
    - You place plates on top → PUSH
    - You remove from top → POP
2. **Undo Feature in Software** 💻
    - Last action is undone first
3. **Books Stack** 📚
    - You can only take the top book

Examples

1. Pushing numbers:
2. PUSH(10), PUSH(20), PUSH(30)
3. Stack: [10, 20, 30] (top = 30)
4. Popping:
5. POP → removes 30
6. Stack: [10, 20]
7. Expression evaluation:
8. (2 + 3) * 5

2. The STL Stack (C++ Standard Library)

C++ provides a built-in stack using:

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

Basic Operations in STL Stack

Operation	Description
push(x)	Insert element
pop()	Remove top
top()	Access top element
empty()	Check if empty

Example Code

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

int main() {
   stack<int> s;

   s.push(10);
   s.push(20);
   s.push(30);

   cout << "Top: " << s.top() << endl;

   s.pop();

   cout << "Top after pop: " << s.top() << endl;

   return 0;
}
```

Real Use Cases

1. Browser history navigation
2. Function call stack (recursion)
3. Syntax parsing in compilers

3. C++ Stack Interface (Custom Design)

When implementing manually, we define an interface like:

```
class Stack {
```

```
public:
   void push(int x);
   void pop();
   int top();
   bool isEmpty();
};
```

Purpose of Interface

- Defines **what operations exist**
- Hides **how they are implemented**

Examples

1. Banking system transaction stack
2. Undo system in editor
3. Game move history

4. Array-Based Stack Implementation

We use:

- Array to store elements
- Integer `top` to track position

Structure

```
#define MAX 100

class Stack {
   int arr[MAX];
   int top;

public:
   Stack() {
      top = -1;
   }
};
```

Explanation

- `top = -1` → stack is empty
- When pushing → increment top
- When popping → decrement top

5. PUSH Operation

Definition

Insert an element into the stack

Condition

- Check **Overflow** (stack full)

Syntax

```
void push(int x) {
   if (top == MAX - 1) {
      cout << "Stack Overflow\n";
      return;
   }
   arr[++top] = x;
}
```

Explanation

- ++top → move to next position
- Store value

Examples

1. PUSH(5), PUSH(10)
2. Adding function calls to stack
3. Saving game progress

Real-Life Analogy

Adding a book on top of a pile 📚

6. POP Operation

Definition

Remove the top element

Condition

- Check **Underflow** (empty stack)

Syntax

```
void pop() {
   if (top == -1) {
      cout << "Stack Underflow\n";
      return;
   }
   top--;
}
```

Explanation

- Just decrease top
- Element is “removed logically”

Examples

1. Undo last action
2. Backspace operation
3. Removing last visited page

Analogy

Taking the top plate 🍽️

7. isEmpty Function

Definition

Checks whether stack is empty

Syntax

```
bool isEmpty() {
   return (top == -1);
}
```

Examples

1. Checking if undo history exists
2. Checking call stack before return
3. Verifying empty container

8. Complete Stack Implementation (C++)

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

```
#define MAX 100

class Stack {
    int arr[MAX];
    int top;

public:
    Stack() {
        top = -1;
    }

    void push(int x) {
        if (top == MAX - 1) {
            cout << "Stack Overflow\n";
            return;
        }
        arr[++top] = x;
    }

    void pop() {
        if (top == -1) {
            cout << "Stack Underflow\n";
            return;
        }
        top--;
    }

    int peek() {
        if (top == -1) {
            cout << "Stack is Empty\n";
            return -1;
        }
        return arr[top];
    }

    bool isEmpty() {
        return (top == -1);
    }
};

int main() {
    Stack s;

    s.push(10);
    s.push(20);
    s.push(30);

    cout << "Top element: " << s.peek() << endl;

    s.pop();

    cout << "Top after pop: " << s.peek() << endl;
```

```
    return 0;
}
```

9. Time Complexity

Operation	Complexity
PUSH	O(1)
POP	O(1)
TOP	O(1)
isEmpty	O(1)

10. Advantages of Stack

✔ Simple and fast
✔ Useful in recursion
✔ Efficient memory usage

11. Limitations

❌ Fixed size (array version)
❌ No random access
❌ Overflow possible

12. Where Stacks Are Used (Important for Exams)

1. Expression Evaluation
2. Parentheses Checking
3. Backtracking Algorithms
4. Depth First Search (DFS)
5. Recursion Call Stack

Types of Stacks Based on Behavior (Conceptual Types)

1. Simple Stack (Linear Stack)

Normal stack (LIFO)

- Push → insert at top
- Pop → remove from top

Example:

Push: 10, 20, 30
Stack: [10, 20, 30]
Pop → removes 30

2. Dynamic Stack

Stack that can grow/shrink dynamically

- Implemented using:
  - Linked list ☑
  - Dynamic array ☑

3. Multiple Stack

More than one stack in a single array

```
int arr[10];
```

- Stack1 → left side
- Stack2 → right side

Use Case:

- Memory optimization

4. Circular Stack (Rare Concept)

Like circular queue but applied to stack logic
(Not commonly used in practice)

5. Recursive Stack (Call Stack)

Used internally by functions

Example:

```
void func() {
   func();
}
```

- Each function call is pushed to call stack
- Managed automatically by compiler

Final Summary

- Stack = LIFO structure
- Main operations:
  - PUSH → insert
  - POP → remove
  - TOP → view element
  - isEmpty → check state
- Can be implemented using:
  - Arrays
  - Linked Lists
  - STL (ready-made)