Stack:

A stack is a linear data structure in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack. It works on LIFO (last in, first out) principle.

Types of Stacks Implementation:

Using Array:

As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs. This type of stack is usually implemented through an array.

Using Linked List:

A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

However, today we will implement both, but the size of the stack will be fixed. We will not grow it dynamically.

Stack Implementation Using an Array

1. The following code defines a simple stack data structure using a class named Stack. Inside the class, you have three private member variables: top (indicating the index of the top element in the stack), size (representing the maximum size of the stack), and arr (a dynamically allocated integer array to store stack elements).
2. Constructor Stack(int size): This constructor initializes the stack with the given size. It sets top to -1 and dynamically allocates an array of integers with a size specified during object creation.
3. isEmpty(): This method checks if the stack is empty by inspecting whether top is -1. It prints a message if the stack is empty and returns true, otherwise, it returns false.
4. isFull(): This method checks if the stack is full by comparing top to the maximum size minus one. It prints a message if the stack is full and returns true, otherwise, it returns false.
5. push(int val): This method adds an element val to the top of the stack if the stack is not full. It increments top to the next position and assigns val to that position in the array. It also prints a message indicating the element pushed.
6. pop(): This method removes the top element from the stack if the stack is not empty. It prints the element being removed and decrements top to point to the new top.
7. peek(): This method allows you to view the top element of the stack without removing it. It simply prints the value at the top of the stack.

8. display(): This method displays all the elements in the stack, from the top to the bottom. It first checks if the stack is not empty and then iterates through the elements in reverse order, printing each one.

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

class Stack
{
	int top;
	int size;
	int *arr;

	public:
	Stack(int size)
	{
		top=-1;
		this->size=size;
		arr=new int[size];
	}
	~Stack()
	{
		delete[] arr;
	}

	bool isEmpty()
	{
		if(top==-1)
		{
			cout<<"Stack is empty"<<endl;
			return true;
		}
		else
		{
			return false;
		}
	}

	bool isFull()
	{
		if(top==size-1)
		{
			cout<<"Stack is full"<<endl;
			return true;
		}
		else
		{
			return false;
		}
	}
```

```
void push(int val)
	{
		if(!isFull())
		{
			top++;
			arr[top]=val;
			cout<<"Element: "<<val<<" is pushed"<<endl;
		}
	}

	void pop()
	{
		if(!isEmpty())
		{
			cout<<"Element: "<<arr[top]<<" is popped"<<endl;
			top--;
		}
	}

	void peek()
	{
		If(!isEmpty())
		{
			cout<<"Peeking into the stack (accessing top element)"<<endl;
			cout<<arr[top]<<endl;
		}
	}

	void display()
	{
		if(!isEmpty())
		{
			cout<<"Stack elements: "<<endl;
			for(int i=top; i>=0; i--)
			{
				cout<<arr[i]<<endl;
			}
			cout<<endl;
		}
	}
};
```

Output:

```
Element: 2 is pushed
Stack elements:
2
Element: 3 is pushed
Stack elements:
3
2
Element: 4 is pushed
Stack elements:
4
3
2
Element: 4 is popped
Stack elements:
3
2
Element: 8 is pushed
Stack elements:
8
3
2
Peeking into the stack (accessing top element)
8
```

Stack using Single Linked List:

1. **Node Class:** Represents nodes in a singly linked list. Contains val to store values and next to point to the next node. Constructor initializes val and sets next to NULL.
2. **Stack Class:** Implements a stack using a singly linked list. Maintains top as a pointer to the top node, limit for the maximum size, and length for the current size.
3. **push(int val):** Adds a new node with val to the top of the stack if not full. Creates a new node, links it to the current top, updates top, and increments length.
4. **pop():** Removes the top node if not empty. Updates top, deletes the old top node for memory management, and decrements length.
5. **Other Functions:**
   - isEmpty() checks if the stack is empty.
   - isFull() checks if the stack is full.
   - peek() allows viewing the top element.
   - display() prints all stack elements from top to bottom. Here, we have implemented the display method using a simple loop that iterates through the top. However, it is possible to display stack like this, but it violates the rules of stack. So, an alternate approach can be you should pop and display the elements from the stack. If you want to retain those elements, pop and save those elements in another data structure, e.g. another stack. Then after displaying the elements push back the elements from the temporary stack to the actual stack.
6. **Main Function:** Demonstrates stack operations: pushing, popping, peeking, and displaying elements within the stack.

```
#include<iostream>
using namespace std;
class Node
{
	public:
	int val;
	Node *next;
	Node(int val)
	{
		this->val=val;
		next=NULL;
	}
};
class Stack
{
	Node *top;
	int limit;
	int length;

	public:
	Stack(int limit)
	{
		this->limit=limit;
		top=NULL;
		length=0;
	}
	~Stack() {
    while (!isEmpty()) {
      pop();
    }
  }
	bool isEmpty()
	{
		if(top==NULL)
		{
			return true;
			cout<<"Stack is empty"<<endl;
		}
		else
		{
			return false;
		}
	}
```

```
bool isFull()
	{
		if(length==limit)
		{
			cout<<"Stack is full"<<endl;
			return true;
		}
		else
		{
			return false;
		}
	}
	void push(int val)
	{
		if(!isFull())
		{
			Node *temp=new Node(val);
			temp->next=top;
			top=temp;
			length++;
			cout<<"Element: "<<val<<" is pushed"<<endl;
		}
	}
	void pop()
	{
		Node *temp=top;
		int val=top->val;
		top=top->next;
		delete temp;
		temp=NULL;
		length--;
		cout<<"Element: "<<val<<" is popped"<<endl;
	}

	void peek()
	{
		if(!isEmpty())
		{
			cout<<"Top of the stack: "<<top->val<<endl;
		}
	}
```

```
void display()
        {
                if(!isEmpty())
                {
                        cout<<"Stack Elements: "<<endl;
                        Node *temp=top;
                        while(temp!=NULL)
                        {
                                cout<<temp->val<<endl;
                                temp=temp->next;
                        }
                        cout<<endl;
                }
        }
};

int main()
{
        Stack s(5);
        s.push(2);
        s.display();
        s.push(3);
        s.display();
        s.push(4);
        s.display();

        s.pop();
        s.display();

        s.push(8);
        s.display();

        s.peek();
}
```

Output:

Element: 2 is pushed

Stack elements:

2

Element: 3 is pushed

Stack elements:

3

2

Element: 4 is pushed

Stack elements:

4

3

2

Element: 4 is popped

Stack elements:

3

2

Element: 8 is pushed

Stack elements:

8

3

2

Top of the stack:

8

Tasks:

Task 1:

Write a program to reverse a string using an array based stack. You must push each character of the string onto the stack and then pop characters to form the reversed string.

Test Case: Input: *hello*
Output: *olleh*

Task 2:

Create a stack that supports push(), pop(), and retrieving the minimum element using linked list.

1. **Create a class MinStack**:
    - Maintain a secondary stack that tracks the minimum value at each level.

Tasks 2.1:

- Implement the MinStack class.
- Test with a series of push and pop operations, and retrieve the minimum value after each operation.