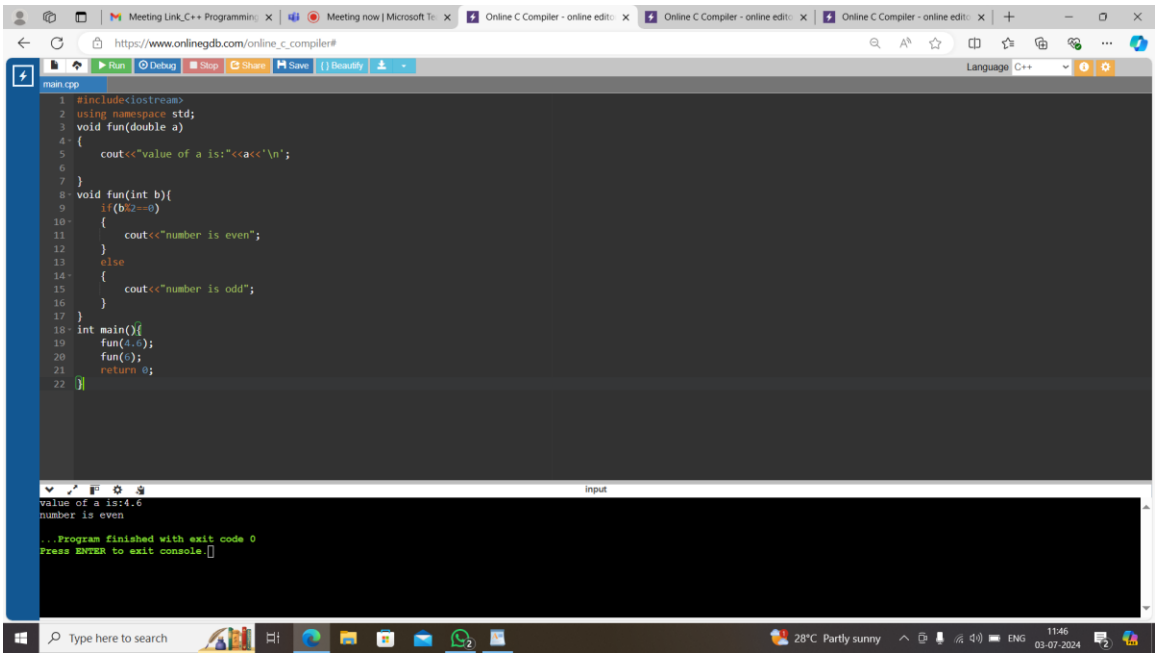
CLASS TEMPLATE :-

```
#include<iostream>
using namespace std;
void fun(double a)
{
	cout<<"value of a is:"<<a<<'\n';
}
void fun(int b){
	if(b%2==0)
	{
		cout<<"number is even";
	}
	else
	{
		cout<<"number is odd";
	}
}
int main(){
	fun(4.6);
	fun(6);
	return 0;
}
```

OUTPUT :-

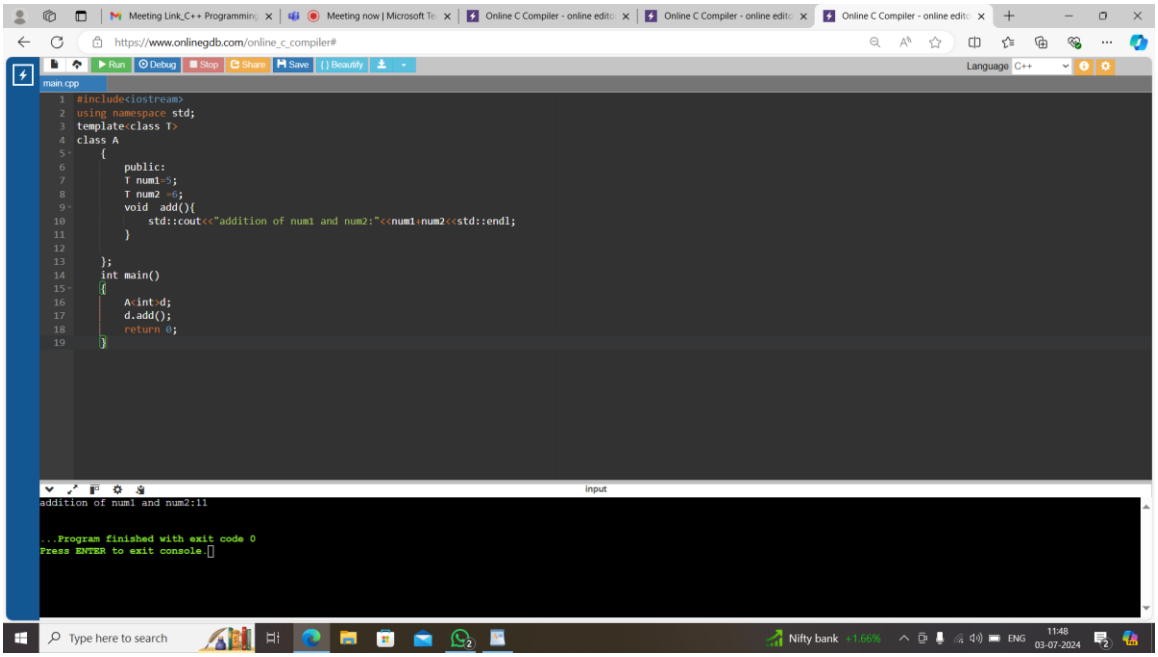


CLASS TEMPLATE WITH ADDITION :-

```
#include<iostream>
using namespace std;
template<class T>
class A
    {
        public:
        T num1=5;
        T num2 =6;
        void    add(){
            std::cout<<"addition of num1 and num2:"<<num1+num2<<std::endl;
        }
    };
    int main()
    {
        A<int>d;
```

```
        d.add();
        return 0;
    }
```

OUTPUT :-

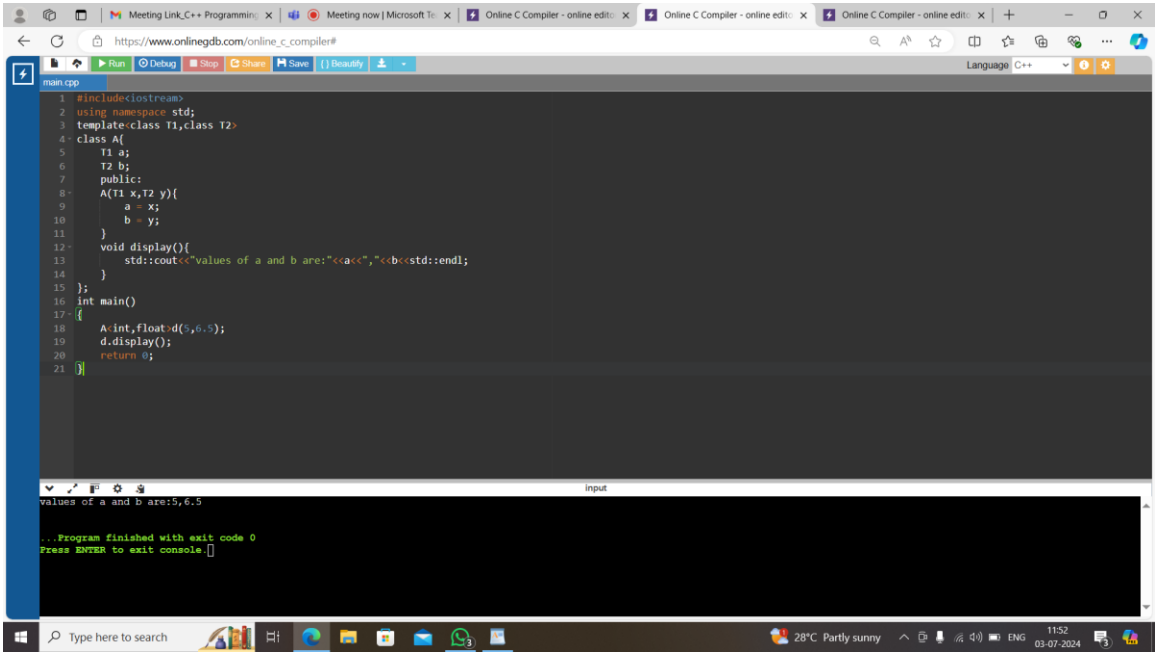


CLASS TEMPLATE WITH MULTIPLE PARAMETERS :-

```
#include<iostream>
using namespace std;
template<class T1,class T2>
class A{
    T1 a;
    T2 b;
    public:
    A(T1 x,T2 y){
        a = x;
        b = y;
    }
```

```
    void display(){
        std::cout<<"values of a and b are:"<<a<<","<<b<<std::endl;
    }
};
int main()
{
    A<int,float>d(5,6.5);
    d.display();
    return 0;
}
```

OUTPUT :-



Design a generic data processing library using class and function templates in C++. This library should be able to handle various data types (e.g., integers, floats, strings) without code duplication.

Requirements:

Create a class template named DataContainer that can hold elements of any data type specified during instantiation.

Implement member functions for DataContainer:

DataContainer(size_t size): Constructor to initialize the container with a specific size.

T& operator[](size_t index): Overloaded subscript operator to access elements.

void printAll(): Prints all elements of the container.

Create a function template named swap that takes two DataContainer objects as arguments and swaps their elements.

Ensure proper memory management using appropriate constructors and destructors.

Implement the DataContainer class template:

Define the template parameter to specify the data type.

Use an array or a vector internally to store the elements.

Implement the constructor, subscript operator, and printAll function as described in the requirements.

Implement the swap function template:

Take two DataContainer objects as arguments.

Use a loop or recursion to iterate over corresponding elements and swap their values.

Consider potential edge cases (e.g., containers of different sizes).

Write a main function to demonstrate the library:

Create instances of DataContainer for different data types (e.g., int, float, string).

Populate the containers with sample data.

Call printAll on each container to verify its contents.

Use the swap function to swap elements between containers of the same type.

Print the containers again to confirm the swap.

Enhance the DataContainer class:

Add member functions for:

size(): Returns the current size of the container.

push_back(const T& value): Appends an element to the back of the container (dynamically resize if necessary).

Modify the constructor to accept an optional initial size (default to 0).

Explore advanced functionalities (optional):

Implement a class template for linked lists or binary search trees, leveraging the DataContainer class.

Create function templates for generic sorting algorithms (e.g., bubble sort, selection sort).

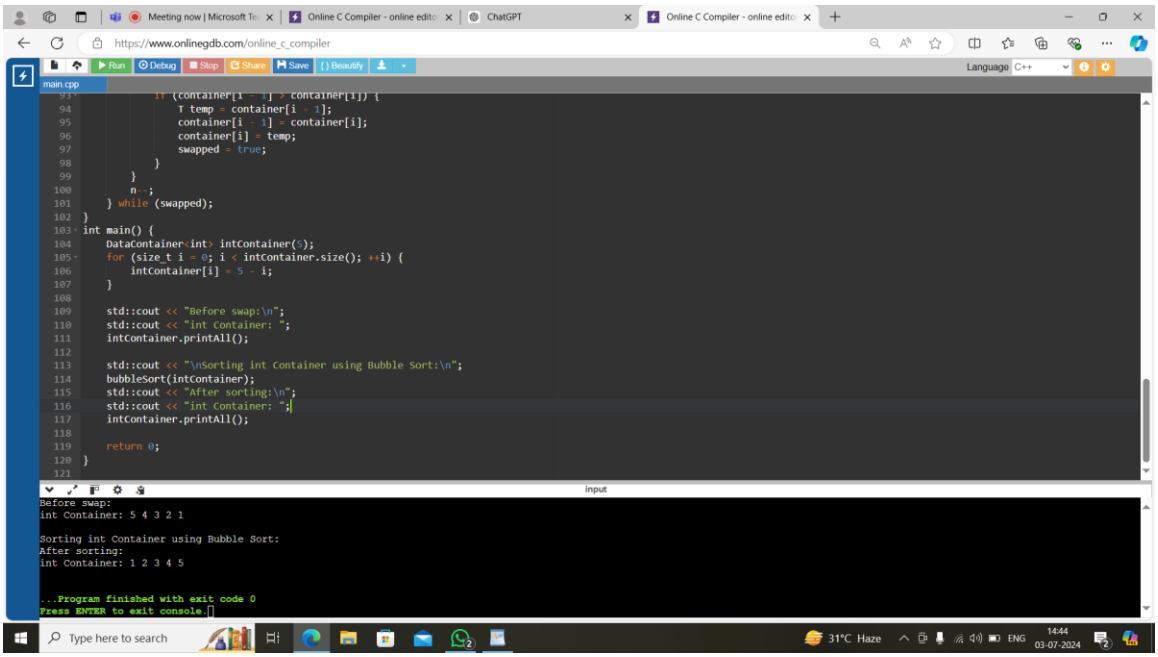
```
 #include <iostream>
#include <string>
template <typename T>
class DataContainer {
private:
    T* elements;
    size_t capacity;
    size_t currentSize;
public:
    DataContainer(size_t size = 0) : capacity(size), currentSize(size) {
        elements = new T[capacity];
    }
    ~DataContainer() {
        delete[] elements;
    }
    DataContainer(const DataContainer& other) : capacity(other.capacity),
currentSize(other.currentSize) {
        elements = new T[capacity];
        for (size_t i = 0; i < currentSize; ++i) {
            elements[i] = other.elements[i];
        }
    }
    DataContainer& operator=(const DataContainer& other) {
        if (this != &other) {
            delete[] elements;
            capacity = other.capacity;
```

```
            currentSize = other.currentSize;
            elements = new T[capacity];
            for (size_t i = 0; i < currentSize; ++i) {
                elements[i] = other.elements[i];
            }
        }
        return *this;
}

T& operator[](size_t index) {
        if (index >= currentSize) {
            std::cerr << "Error: Index out of range" << std::endl;
            exit(1);
        }
        return elements[index];
}
void printAll() const {
        for (size_t i = 0; i < currentSize; ++i) {
            std::cout << elements[i] << " ";
        }
        std::cout << std::endl;
}
size_t size() const {
        return currentSize;
}
void push_back(const T& value) {
        if (currentSize >= capacity) {
            size_t newCapacity = capacity == 0 ? 1 : capacity * 2;
```

```
            T* newElements = new T[newCapacity];
            for (size_t i = 0; i < currentSize; ++i) {
                newElements[i] = elements[i];
            }
            delete[] elements;
            elements = newElements;
            capacity = newCapacity;
        }
        elements[currentSize++] = value;
    }
    template <typename U>
    void swap(DataContainer<U>& other) {
        if (currentSize != other.size()) {
            std::cerr << "Error: Containers must be of the same size to swap" << std::endl;
            exit(1);
        }
        for (size_t i = 0; i < currentSize; ++i) {
            T temp = elements[i];
            elements[i] = static_cast<T>(other[i]);
            other[i] = static_cast<U>(temp);
        }
    }
};
template <typename T>
void bubbleSort(DataContainer<T>& container) {
    size_t n = container.size();
    bool swapped;
    do {
```

```
            swapped = false;
            for (size_t i = 1; i < n; ++i) {
                if (container[i - 1] > container[i]) {
                    T temp = container[i - 1];
                    container[i - 1] = container[i];
                    container[i] = temp;
                    swapped = true;
                }
            }
            n--;
        } while (swapped);
}
int main() {
        DataContainer<int> intContainer(5);
        for (size_t i = 0; i < intContainer.size(); ++i) {
            intContainer[i] = 5 - i;
        }
        std::cout << "Before swap:\n";
        std::cout << "int Container: ";
        intContainer.printAll();
        std::cout << "\nSorting int Container using Bubble Sort:\n";
        bubbleSort(intContainer);
        std::cout << "After sorting:\n";
        std::cout << "int Container: ";
        intContainer.printAll();
        return 0;
}
```

OUTPUT :-

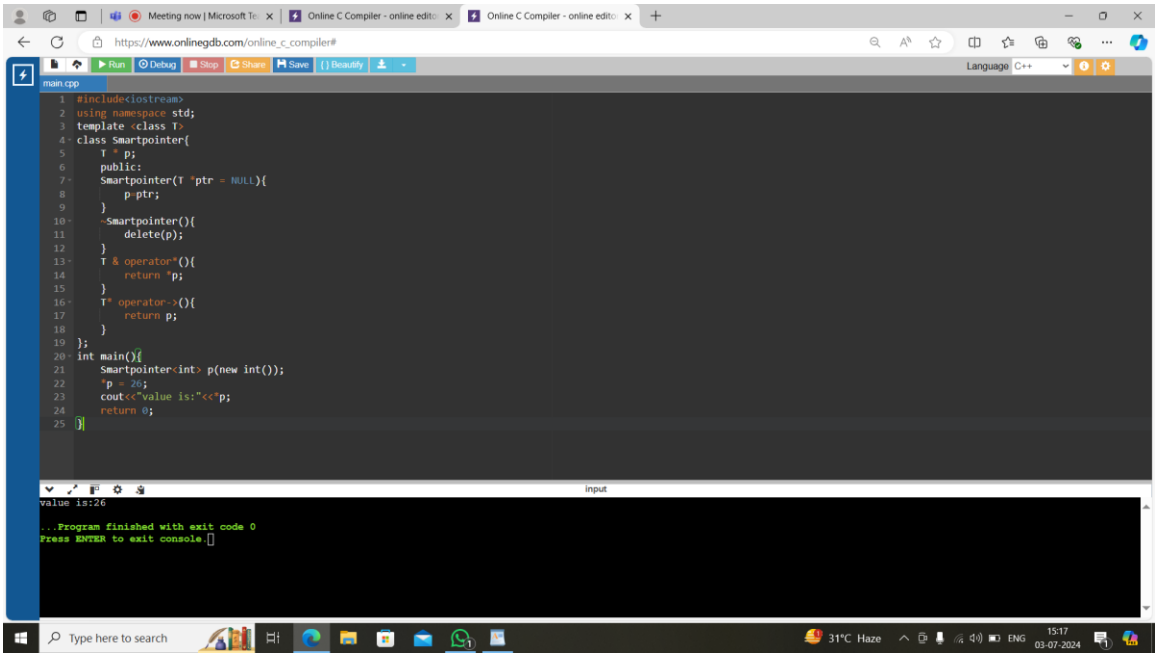


SMART POINTER :-

```
#include<iostream>
using namespace std;
template <class T>
class Smartpointer{
    T * p;
    public:
    Smartpointer(T *ptr = NULL){
        p=ptr;
    }
    ~Smartpointer(){
        delete(p);
    }
    T & operator*(){
        return *p;
    }
```

```
    T* operator->(){
        return p;
    }
};
int main(){
    Smartpointer<int> p(new int());
    *p = 26;
    cout<<"value is:"<<*p;
    return 0;
}
```

OUTPUT :-



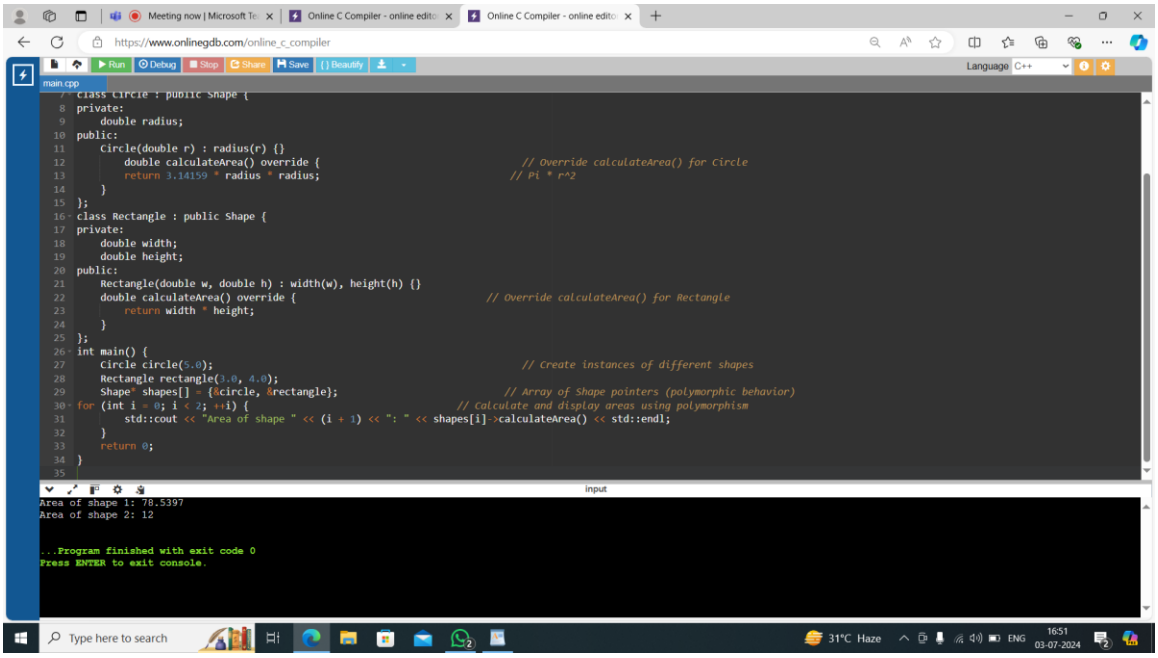
In object-oriented programming with C++, abstract classes are a valuable tool for defining common interfaces and behaviors for a group of related classes. However, directly creating objects from an abstract class is not possible. This problem statement explores how abstract classes are used to enforce a design pattern and promote code reusability.

use abstract classes and polymorphism in C++ for calculating the areas of various shapes

```
#include <iostream>

class Shape {                                   // Abstract base class

public:

    virtual double calculateArea() = 0;                                 // Pure virtual function for
calculating area

        virtual ~Shape() {}                                             // Virtual destructor
(always good practice in polymorphic hierarchies)

};

class Circle : public Shape {

private:

    double radius;

public:

    Circle(double r) : radius(r) {}

        double calculateArea() override {                               // Override
calculateArea() for Circle

        return 3.14159 * radius * radius;                               // Pi * r^2

    }

};

class Rectangle : public Shape {

private:

    double width;

    double height;

public:

    Rectangle(double w, double h) : width(w), height(h) {}

    double calculateArea() override {                                   // Override
calculateArea() for Rectangle

        return width * height;

    }

};
```

```
int main() {
    Circle circle(5.0);                                                    // Create
instances of different shapes
    Rectangle rectangle(3.0, 4.0);
    Shape* shapes[] = {&circle, &rectangle};                               // Array of Shape
pointers (polymorphic behavior)
    for (int i = 0; i < 2; ++i) {                                          // Calculate and display areas
using polymorphism
        std::cout << "Area of shape " << (i + 1) << ": " << shapes[i]->calculateArea() << std::endl;
    }
    return 0;
}
```

OUTPUT :-



Inventory Management System:

Problem: Design a system to manage inventory for various products. Each product might have different attributes (name, price, quantity) and potentially unique functionalities (e.g., perishable items with an expiry date).

File Processing System:

Problem: Develop a system for handling different file formats (text, image, binary). Each format might require specific read/write operations.

```
#include <iostream>
#include <fstream>
#include <string>
class File {                                        // Abstract base class for file operations
protected:
    std::string filename;
public:
    File(const std::string& fname) : filename(fname) {}
    virtual void read() const = 0;
    virtual void write() const = 0;
    virtual ~File() {}
};
class TextFile : public File {                      // Text file class (inherits from File)
private:
    std::string content;
public:
    TextFile(const std::string& fname) : File(fname) {}
    void read() const override {
        std::ifstream file(filename);
        if (file.is_open()) {
            std::string line;
            while (std::getline(file, line)) {
                std::cout << line << std::endl;
```

```
            }
            file.close();
        } else {
            std::cerr << "Unable to open file: " << filename << std::endl;
        }
    }
    void write() const override {
        std::ofstream file(filename);
        if (file.is_open()) {
            file << "Example text content.\n";
            file.close();
        } else {
            std::cerr << "Unable to create file: " << filename << std::endl;
        }
    }
};
class ImageFile : public File {                                    // Image file class (inherits from File)
public:
    ImageFile(const std::string& fname) : File(fname) {}
    void read() const override {
        std::cout << "Reading image file: " << filename << std::endl;                    // Implement
image file reading logic
    }
    void write() const override {
        std::cout << "Writing image file: " << filename << std::endl;                                //
Implement image file writing logic
    }
};
```

```
int main() {
    // Example usage
    TextFile textFile("example.txt");
    textFile.write();
    textFile.read();
    ImageFile imageFile("example.jpg");
    imageFile.write();
    imageFile.read();
    return 0;
}
```

OUTPUT :-