OOP Course Outline

1. Introduction to OOP

- ☐ What is Object-Oriented Programming?
- ☐ Why use OOP? Benefits and real-world analogy
- ☐ Comparison between Procedural and Object-Oriented Programming

2. Basic Concepts of OOP — The Four Pillars

- ☐ **Encapsulation**
  - ☐ What is encapsulation?
  - ☐ Access specifiers: private, public, protected
  - ☐ Getters and setters (accessor and mutator methods)
- ☐ **Inheritance**
  - ☐ What is inheritance?
  - ☐ Types of inheritance: single, multilevel, hierarchical
  - ☐ Using base and derived classes
- ☐ **Polymorphism**
  - ☐ Compile-time polymorphism (Function Overloading, Operator Overloading)
  - ☐ Run-time polymorphism (Virtual functions, Function overriding)

- ☐ **Abstraction**
  - ☐ Abstract classes and interfaces
  - ☐ Hiding complexity and exposing only essentials

3. Classes and Objects

- ☐ Defining classes and creating objects
- ☐ Constructors and Destructors
- ☐ Static members (variables and functions)
- ☐ this pointer

4. Advanced OOP Concepts

- ☐ Friend functions and friend classes
- ☐ Copy constructor and assignment operator overloading
- ☐ Dynamic memory allocation inside classes (new and delete)
- ☐ Exception handling in OOP

5. Function Overloading and Operator Overloading

- ☐ Function overloading
- ☐ Operator overloading: syntax and examples (like +, -, <<, >>)

6. Inheritance Deep Dive

- ☐ Types of inheritance with examples
- ☐ Access control in inheritance
- ☐ Constructor and destructor calls in inheritance
- ☐ Virtual functions and run-time polymorphism
- ☐ Pure virtual functions and abstract classes

7. Templates and Generic Programming (optional but useful)

- ☐ Function templates
- ☐ Class templates

8. File Handling with Classes

- ☐ Reading from and writing to files using OOP
- ☐ File streams and file classes

9. Real-world OOP Design Examples

- ☐ Designing simple real-world systems (e.g., library management, bank accounts)
- ☐ UML basics (class diagrams, relationships)

10. Practice Projects and Assignments

- ☐ Build small OOP-based programs combining multiple concepts
- ☐ Debugging and code organization best practices

What is OOP?

Object-Oriented Programming (OOP) is a way of programming where we **organize code using "objects" and "classes"**.
It helps make programs **clean, reusable, and easy to understand**.

🔑 4 Pillars of OOP (Main Concepts):

1. **Class & Object**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**

Step 1: Class & Object

📦 What is a Class?

A **class** is like a **blueprint** or design.
Example: A class called `Car` describes what a car has (like color, speed) and what it can do (like drive, stop).

🚗 What is an Object?

An **object** is the **actual car made from that blueprint**.
We can create many objects from one class.

Example in C++:

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

class Car {
public:
```

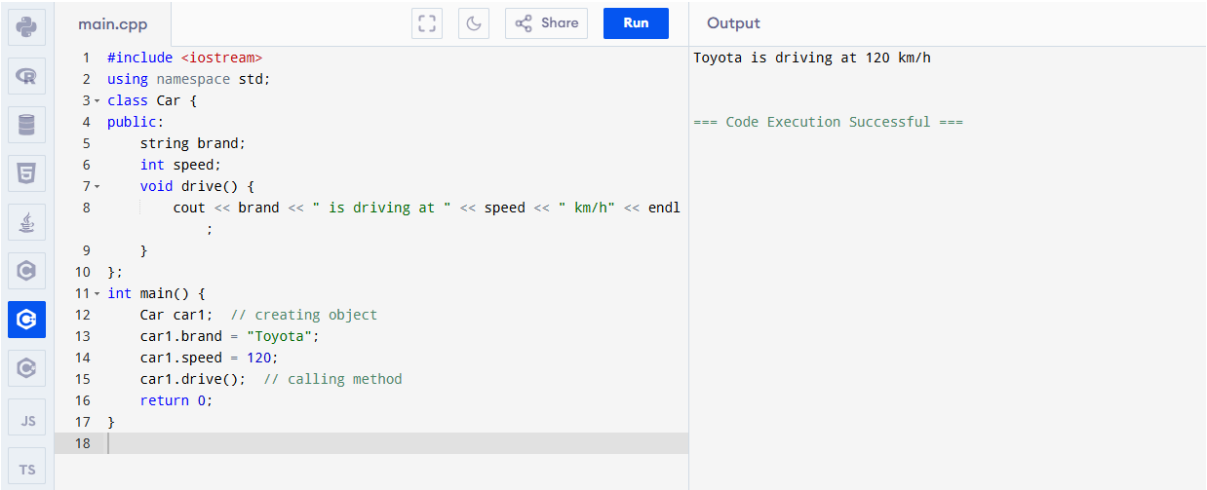
```
    string brand;
    int speed;

    void drive() {
        cout << brand << " is driving at " << speed << " km/h" << endl;
    }
};

int main() {
    Car car1;  // creating object
    car1.brand = "Toyota";
    car1.speed = 120;

    car1.drive();  // calling method

    return 0;
}
```

In this Program:
`class Car` is a blueprint.

`car1` is an object of that class.

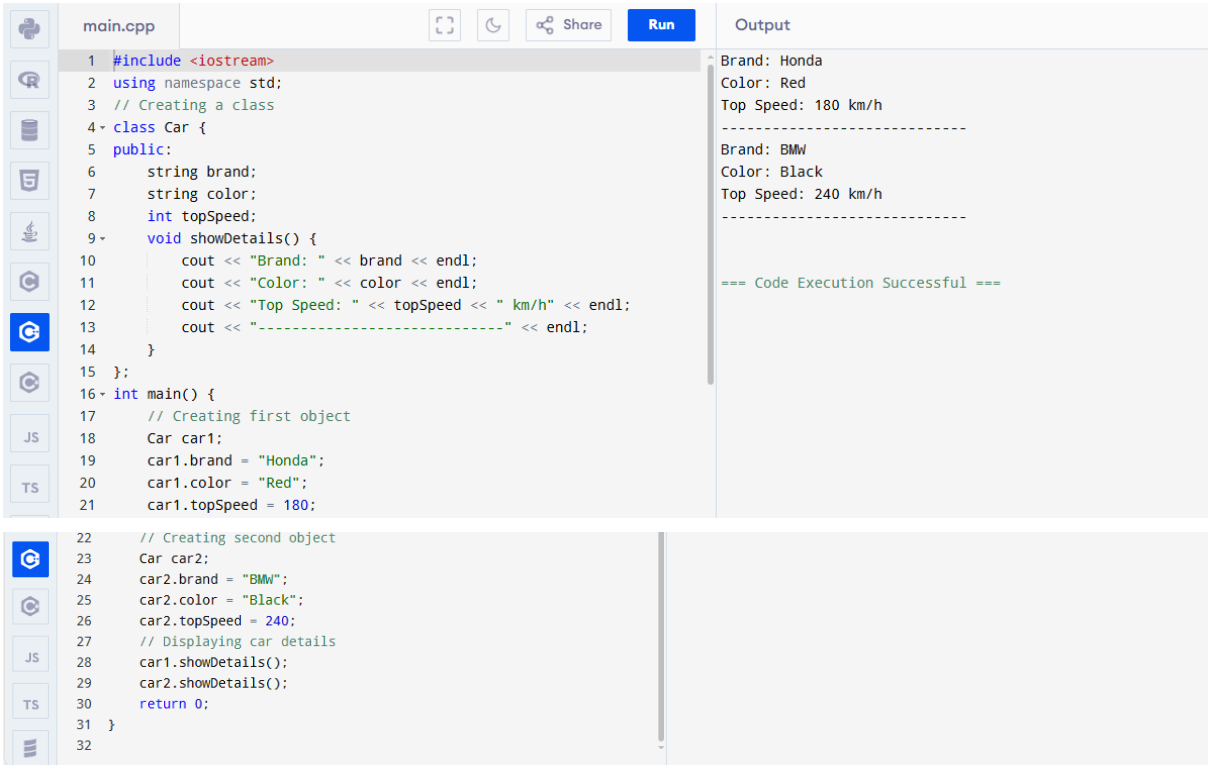
`brand` and `speed` are **data members**.

`drive()` is a **member function** (method).



🚗 Program – Two Cars Using Class

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

// Creating a class
class Car {
public:
    string brand;
    string color;
    int topSpeed;

    void showDetails() {
        cout << "Brand: " << brand << endl;
        cout << "Color: " << color << endl;
        cout << "Top Speed: " << topSpeed << " km/h" << endl;
        cout << "----------------------------" << endl;
    }
};

int main() {
    // Creating first object
    Car car1;
    car1.brand = "Honda";
    car1.color = "Red";
    car1.topSpeed = 180;

    // Creating second object
    Car car2;
    car2.brand = "BMW";
    car2.color = "Black";
    car2.topSpeed = 240;

    // Displaying car details
    car1.showDetails();
    car2.showDetails();

    return 0;
}
```



🔒 2. Encapsulation (Simple Explanation)

🧠 What is Encapsulation?

> **Encapsulation** means **hiding data** and keeping it safe from being directly accessed or changed by anyone outside the class.

We do this by:

- Making variables **private**
- Providing **public functions** to **get or set** the values

Just like in real life:
🔐 *A bank keeps your money safe, but gives you a card to access it.*

🧪 Example in C++

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

class Student {
private:
   int age;  // this is hidden from outside
```

```
public:
    void setAge(int a) {
        if (a > 0)
            age = a;
        else
            cout << "Invalid age!" << endl;
    }

    int getAge() {
        return age;
    }
};

int main() {
    Student s1;
    s1.setAge(20);         // setting the age using public function
    cout << s1.getAge();   // getting the age using public function

    return 0;
}
```

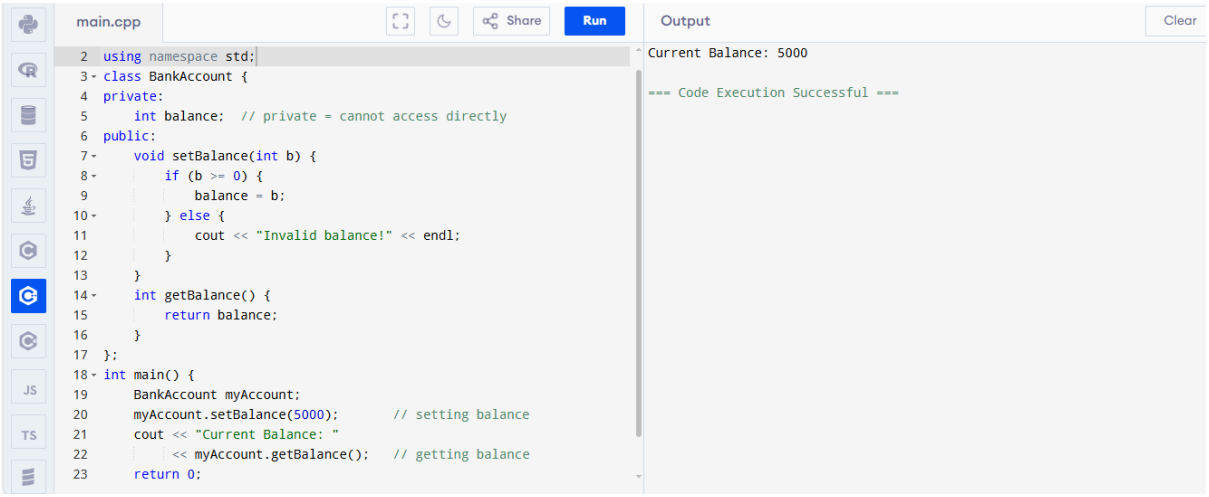
🧩 Key Points:
age is private → cannot access like s1.age = 20;

setAge() and getAge() are public functions → they control access

This makes your code safe and secure = Encapsulation

Practice program for **Encapsulation** using a `BankAccount` class — you'll enter and view account info, but the balance will be **protected** using private data.
Bank Account with Encapsulation

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

class BankAccount {
private:
    int balance;  // private = cannot access directly

public:
    void setBalance(int b) {
        if (b >= 0) {
            balance = b;
        } else {
            cout << "Invalid balance!" << endl;
        }
    }

    int getBalance() {
        return balance;
    }
};

int main() {
    BankAccount myAccount;

    myAccount.setBalance(5000);       // setting balance
    cout << "Current Balance: "
         << myAccount.getBalance();   // getting balance

    return 0;
}
```



👨‍👦 3. Inheritance (Simple Explanation)

🧠 What is Inheritance?

> **Inheritance** means one class (child) can **use properties and functions** of another class (parent).
> It helps **reuse code** and **build relationships** between classes.

👪 Example:

- Parent Class: `Animal` → can eat, sleep
- Child Class: `Dog` → can also bark + everything Animal can do

📄 Syntax:

```
class Parent {
   // parent code
};

class Child : public Parent {
   // child code
};
```

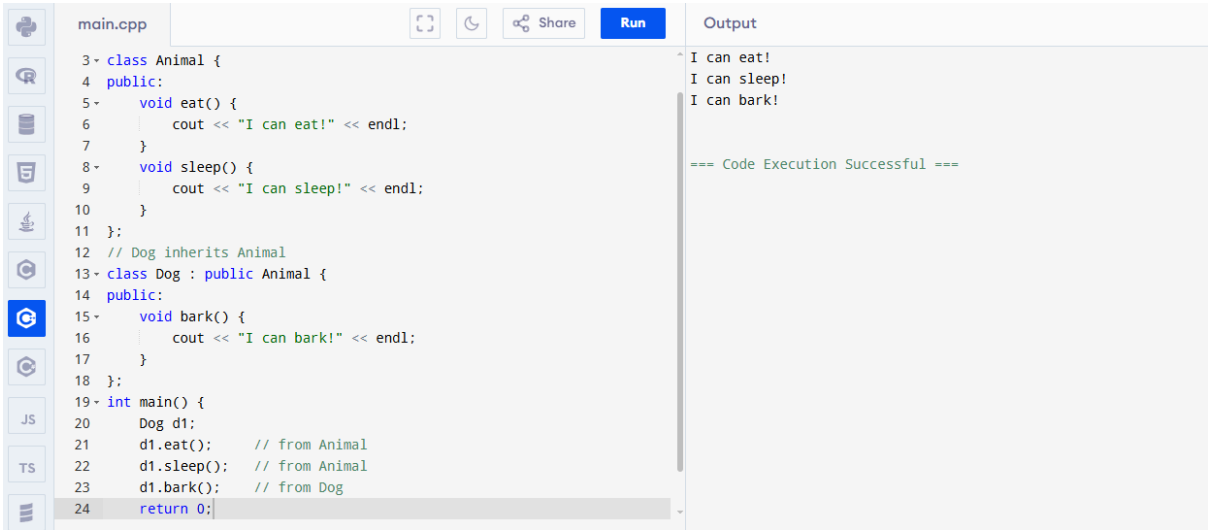
🧪 Example in C++

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

class Animal {
public:
    void eat() {
        cout << "I can eat!" << endl;
    }

    void sleep() {
        cout << "I can sleep!" << endl;
    }
};

// Dog inherits Animal
class Dog : public Animal {
public:
    void bark() {
        cout << "I can bark!" << endl;
    }
};

int main() {
    Dog d1;
    d1.eat();     // from Animal
    d1.sleep();   // from Animal
    d1.bark();    // from Dog

    return 0;
}
```

Important Points:
`Dog` **inherits** from `Animal` using `: public Animal`

Now `Dog` has **all functions** of `Animal`

You can **add extra features** in child class (like `bark()`)



Practice program for **Inheritance** using a real-world example:
We'll make a **`Person`** class and a **`Student`** class that inherits from it.

Program – Inheritance (Person → Student)

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

// Base class (Parent)
class Person {
public:
    string name;
    int age;

    void introduce() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

// Derived class (Child)
class Student : public Person {
public:
    int rollNo;

    void showStudentInfo() {
        cout << "Roll Number: " << rollNo << endl;
    }
};
```

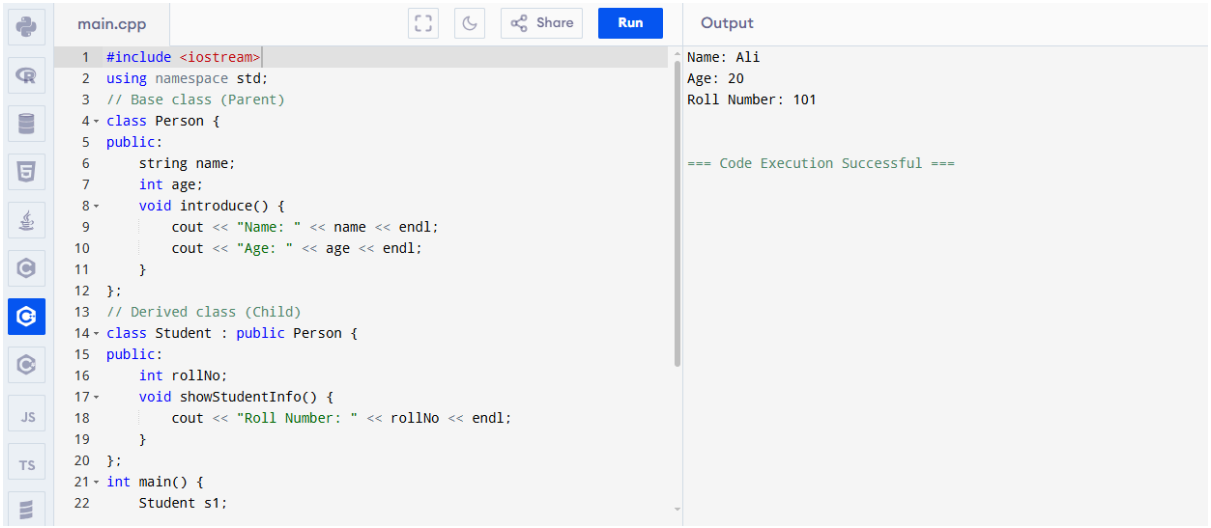
```
int main() {
    Student s1;

    // Accessing parent class members
    s1.name = "Ali";
    s1.age = 20;

    // Accessing child class member
    s1.rollNo = 101;

    s1.introduce();         // from Person class
    s1.showStudentInfo();   // from Student class

    return 0;
}
```

Important Points:

Created a **base class** `Person`

Created a **derived class** `Student` that inherits `Person`

Used **both base and derived class features**



🌀 4. Polymorphism

What is Polymorphism?

> **Polymorphism** means **one name, many forms**.
> A single function or method behaves **differently** based on the **object or parameters**.

There are **two types**:

Type	Example
Compile-Time	Function Overloading
Run-Time	Function Overriding (via Inheritance)

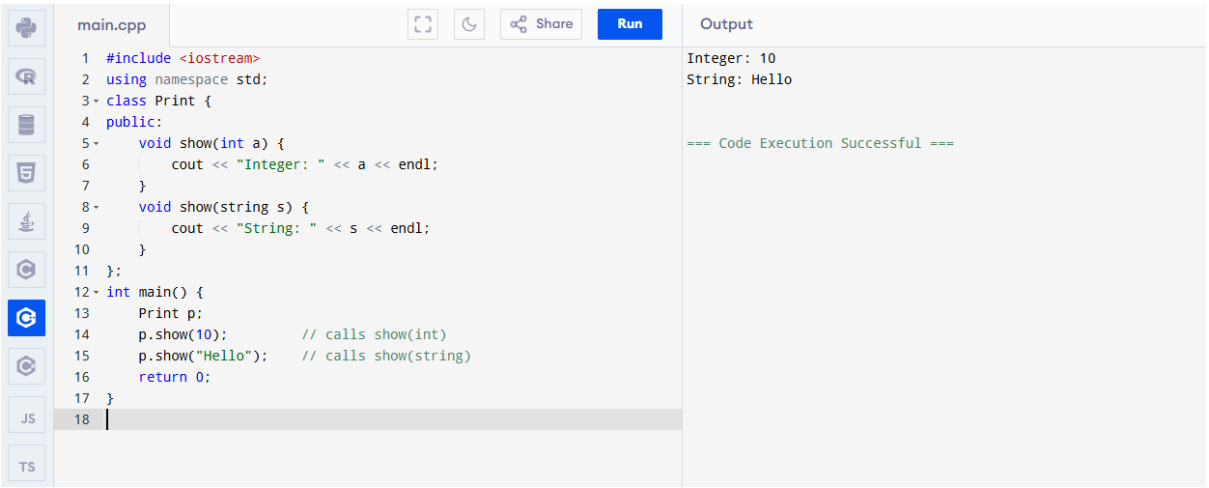
Type 1: Compile-Time Polymorphism

(Function Overloading)

```
#include <iostream>

using namespace std;


class Print {

public:

    void show(int a) {

        cout << "Integer: " << a << endl;

    }
```

```
    void show(string s) {
        cout << "String: " << s << endl;
    }
};

int main() {
    Print p;
    p.show(10);         // calls show(int)
    p.show("Hello");    // calls show(string)

    return 0;
}
```

Note: One function name `show()` behaves differently depending on parameters.

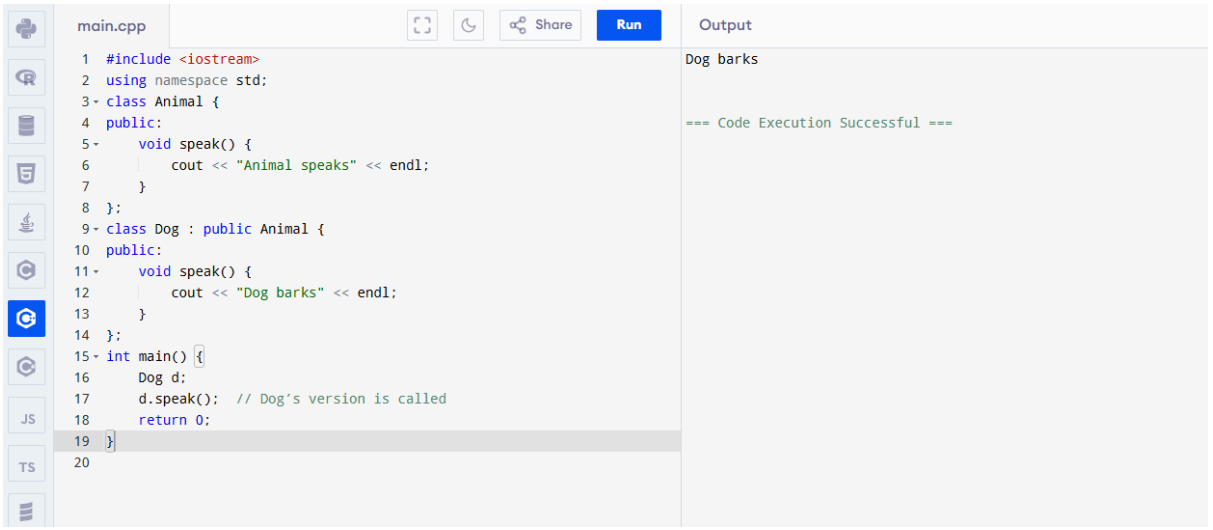


Type 2: Run-Time Polymorphism (Function Overriding)

Example:

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

class Animal {
public:
    void speak() {
        cout << "Animal speaks" << endl;
    }
};

class Dog : public Animal {
public:
    void speak() {
        cout << "Dog barks" << endl;
    }
};

int main() {
    Dog d;
    d.speak();  // Dog's version is called

    return 0;
}
```

Note: Child class **overrides** parent function with the same name.

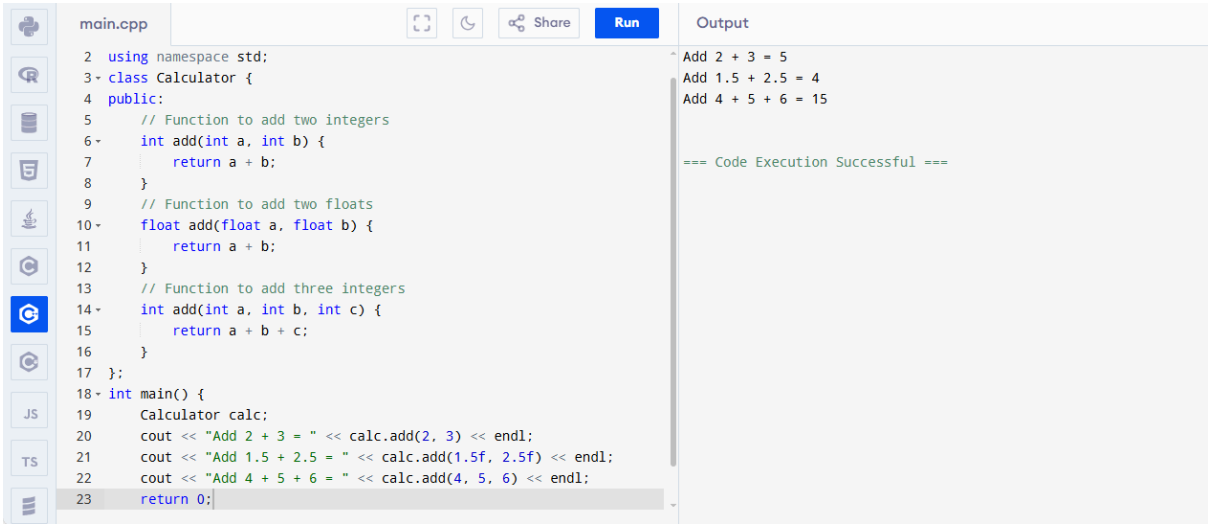


Note: Use `virtual` keyword to make it more flexible at runtime

Program for **Polymorphism**, showing **both Overloading and Overriding** in action — using a `Calculator` example and an `Animal` example.

🔢 Part 1: Compile-Time Polymorphism (Function Overloading)

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

class Calculator {
public:
    // Function to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Function to add two floats
    float add(float a, float b) {
        return a + b;
    }

    // Function to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }
};

int main() {
    Calculator calc;

    cout << "Add 2 + 3 = " << calc.add(2, 3) << endl;
    cout << "Add 1.5 + 2.5 = " << calc.add(1.5f, 2.5f) << endl;
    cout << "Add 4 + 5 + 6 = " << calc.add(4, 5, 6) << endl;

    return 0;
}
```

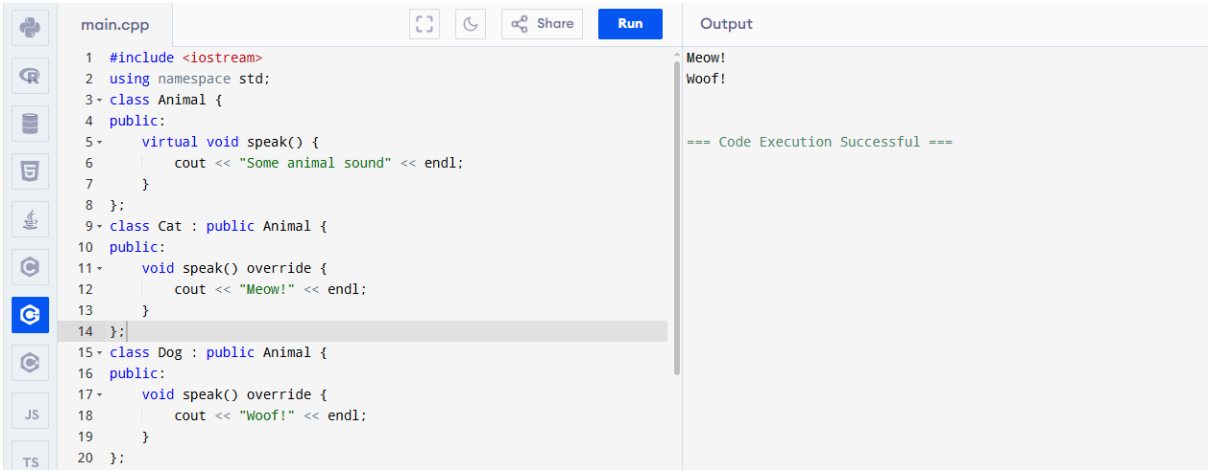


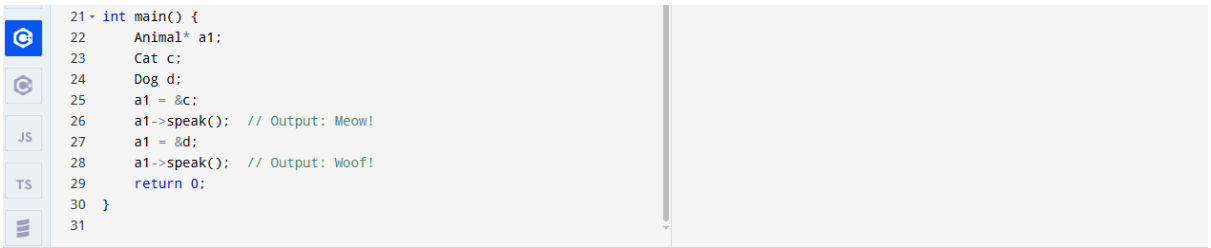
🐶 Part 2: Run-Time Polymorphism

(Function Overriding)

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

class Animal {
public:
    virtual void speak() {
        cout << "Some animal sound" << endl;
    }
};

class Cat : public Animal {
public:
    void speak() override {
        cout << "Meow!" << endl;
    }
};

class Dog : public Animal {
public:
    void speak() override {
        cout << "Woof!" << endl;
    }
};

int main() {
    Animal* a1;
    Cat c;
```

```
    Dog d;

    a1 = &c;
    a1->speak();  // Output: Meow!

    a1 = &d;
    a1->speak();  // Output: Woof!

    return 0;
}
```





🔒 1. Advanced Encapsulation

Encapsulation = Data Protection + Access Control

Concepts:

- **Private members** are hidden from outside the class
- Use **getters** and **setters** to **control access**
- Prevents **unauthorized modification**

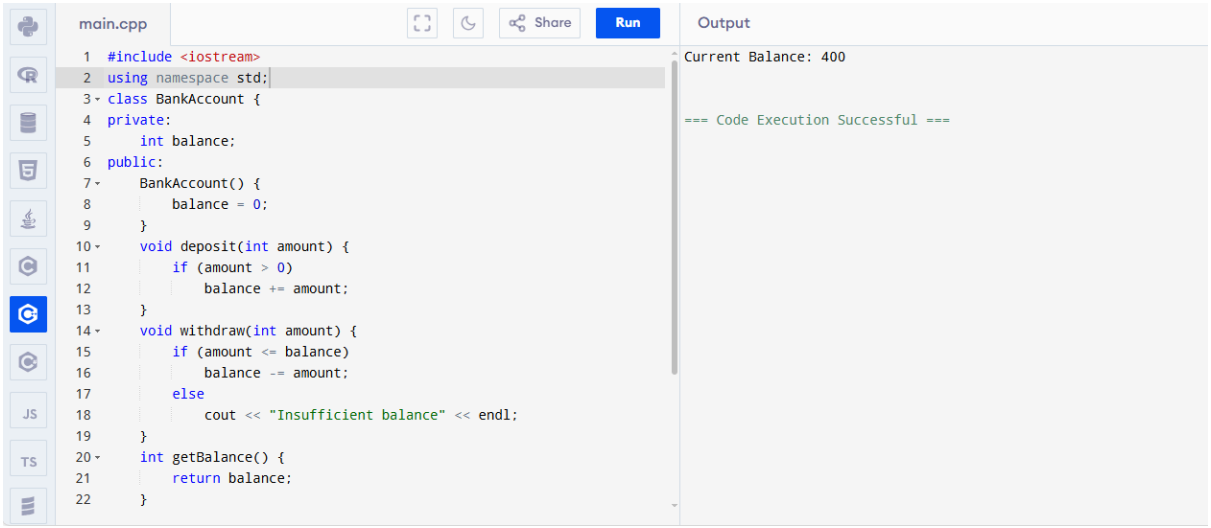
- Makes classes **modular** and **secure**

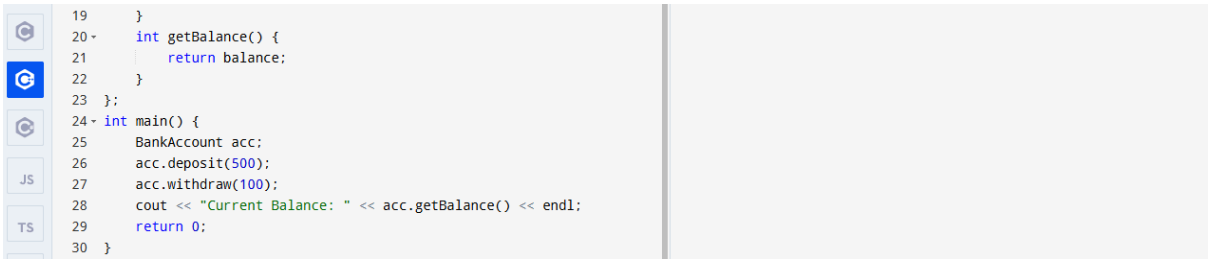
📌 Example:

Bank account class where **balance** is private, and you use **functions** to deposit or withdraw.

🧪 Example:

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

class BankAccount {
private:
    int balance;

public:
    BankAccount() {
        balance = 0;
    }

    void deposit(int amount) {
        if (amount > 0)
            balance += amount;
    }

    void withdraw(int amount) {
        if (amount <= balance)
            balance -= amount;
        else
            cout << "Insufficient balance" << endl;
    }

    int getBalance() {
        return balance;
    }
};

int main() {
    BankAccount acc;
    acc.deposit(500);
    acc.withdraw(100);
    cout << "Current Balance: " << acc.getBalance() << endl;
    return 0;
}
```





👨‍👦 2. Advanced Inheritance

Concepts:

- Inherit **constructors**
- **Protected** access modifier
- **Multilevel**, **Multiple**, **Hybrid** inheritance
- Use **virtual base class** to avoid duplication

🧪 Multilevel Example:

```
class A {
public:
   void funcA() { cout << "A\n"; }
};

class B : public A {
public:
   void funcB() { cout << "B\n"; }
};
```

```
class C : public B {
public:
    void funcC() { cout << "C\n"; }
};
```

🌀 3. Advanced Polymorphism

Concepts:

- **Virtual functions**
- **Abstract classes**
- **Pure virtual functions**
- **Dynamic binding** using pointers

🧪 Abstract Class Example:

```
class Shape {
public:
    virtual void area() = 0; // Pure virtual
};

class Circle : public Shape {
public:
    void area() override {
        cout << "Circle area formula\n";
    }
};

int main() {
    Shape* s = new Circle();
    s->area();
}
```

🧊 4. Advanced Abstraction

Concepts:

- ☐ Use **interfaces** (abstract classes) to design **architecture**
- ☐ Only show **essential features**
- ☐ Real-world: ATM machine (you press buttons, don’t see internal wiring)

Simple OOP Concepts

1. Constructor & Destructor

- **Constructor**: Special function that runs when object is created.
- **Destructor**: Runs when object is deleted (used to clean up memory).

```
class Person {
public:
    Person() { cout << "Constructor called\n"; }
    ~Person() { cout << "Destructor called\n"; }
};
```

2. this Pointer

- Refers to the current object.
- Useful when variables and parameters have the same name.

```
class Box {
    int width;
```

```
public:
    void setWidth(int width) {
        this->width = width;  // disambiguates local and member
    }
};
```

3. Static Members

- Belongs to the **class**, not individual objects.
- Shared by all objects.

```
class MyClass {
public:
    static int count;
};
int MyClass::count = 0;
```

4. Friend Function

- A function **outside the class** that can access **private** members.

```
class A {
    int x;
    friend void show(A);
};
```

5. Function Overloading (Compile-time Polymorphism)

- Same function name, different parameters.

```
void add(int a, int b);
void add(float a, float b);
```

6. Operator Overloading (optional but good to know)

- Redefine operators like `+`, `-`, `==` etc.

```
class Complex {
public:
    int real, imag;
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
};
```

1. Constructor & Destructor

What is Constructor?

- It's a **special function** inside a class.
- It runs **automatically** when you create an object.

- Used to **initialize** variables or do setup.

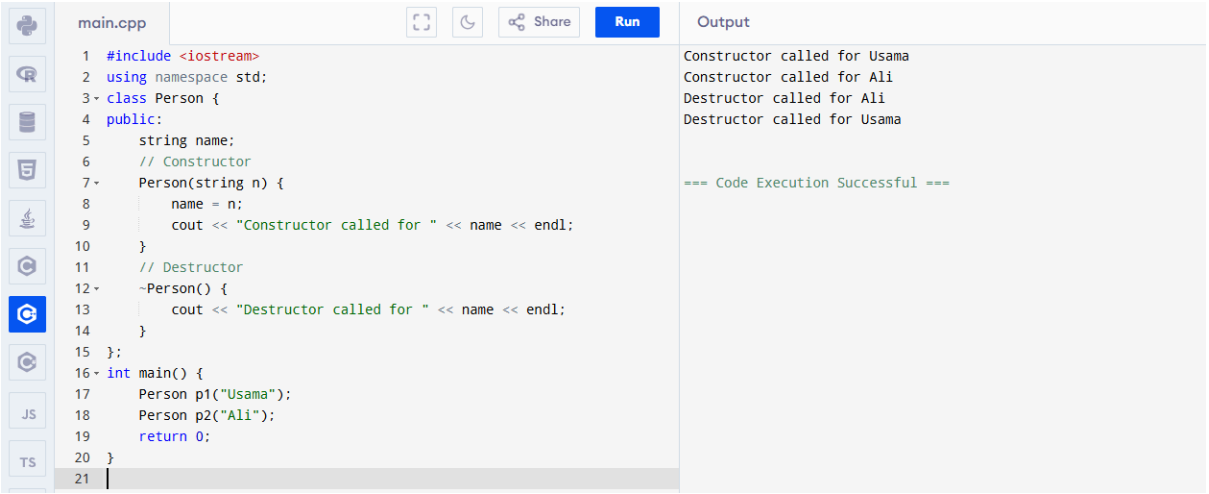
What is Destructor?

- It's a **special function** that runs **when an object is destroyed** or goes out of scope.
- Used to **clean up** resources like memory or files.

Example:

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

class Person {
public:
    string name;

    // Constructor
    Person(string n) {
        name = n;
        cout << "Constructor called for " << name << endl;
    }

    // Destructor
    ~Person() {
        cout << "Destructor called for " << name << endl;
    }
};

int main() {
```

```
    Person p1("Usama");

    Person p2("Ali");

    return 0;

}
```



2. this Pointer

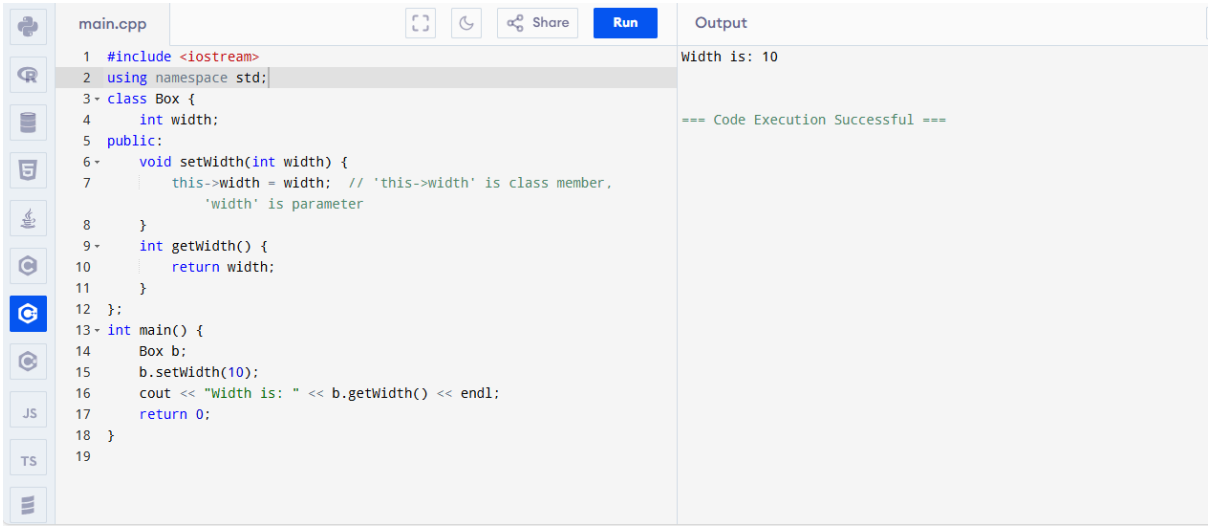
What is `this` pointer?

- Inside a class, `this` is a **pointer** that points to the **current object**.
- Useful to resolve naming conflicts (e.g., when function parameter and class member have same name).

Example:

```
#include <iostream>

using namespace std;


class Box {
```

```
    int width;

public:
    void setWidth(int width) {
        this->width = width;  // 'this->width' is class member, 'width' is parameter
    }

    int getWidth() {
        return width;
    }
};

int main() {
    Box b;
    b.setWidth(10);
    cout << "Width is: " << b.getWidth() << endl;
    return 0;
}
```



3. Static Members

What are static members?

- **Static variables or functions belong to the class**, not to any one object.
- Shared by **all objects** of that class.

Example:

#include <iostream>

using namespace std;

class Counter {

public:

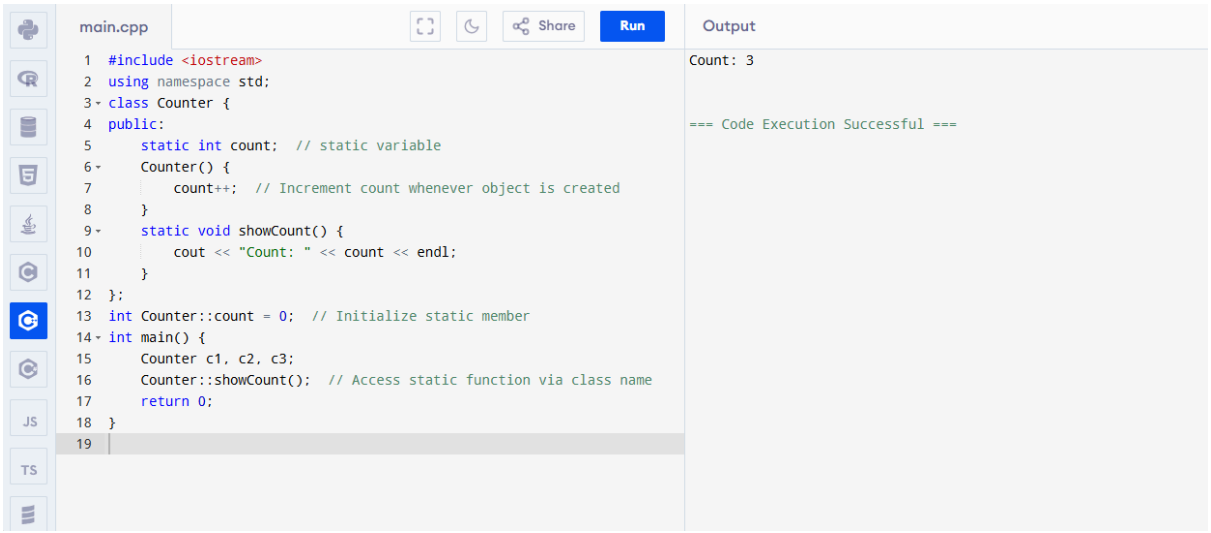
static int count; // static variable

Counter() {

count++; // Increment count whenever object is created

}

```
    static void showCount() {
        cout << "Count: " << count << endl;
    }
};

int Counter::count = 0;  // Initialize static member

int main() {
    Counter c1, c2, c3;
    Counter::showCount();  // Access static function via class name
    return 0;
}
```



4. Friend Function

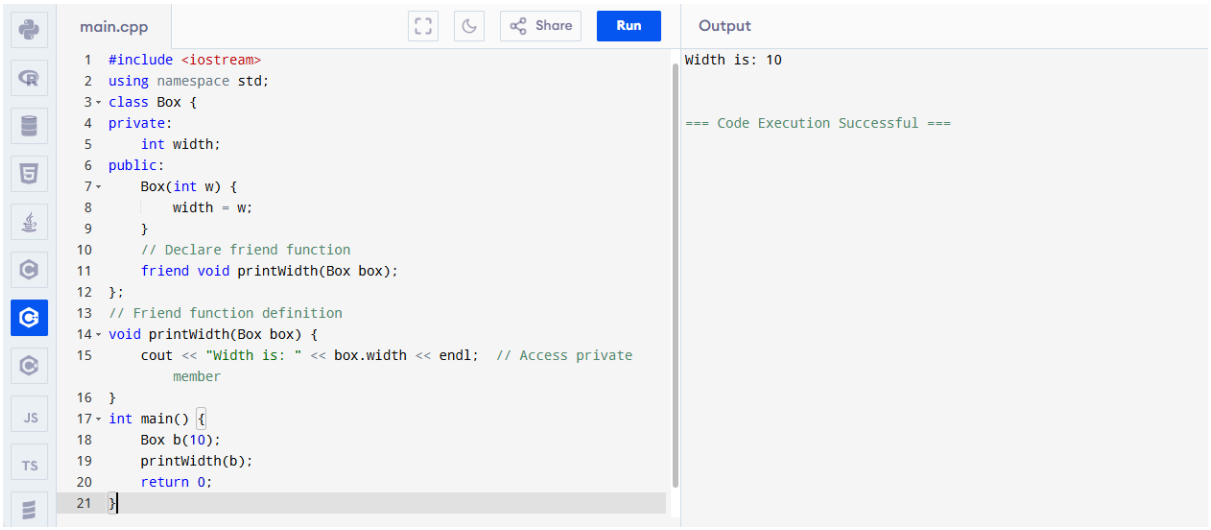
What is a friend function?

- It is a **function outside the class** but can **access private members** of the class.
- Useful when some external function needs special access.

Example:

```
#include <iostream>

using namespace std;


class Box {

private:

    int width;


public:

    Box(int w) {

        width = w;

    }


    // Declare friend function

    friend void printWidth(Box box);

};


// Friend function definition

void printWidth(Box box) {

    cout << "Width is: " << box.width << endl;  // Access private member

}
```

```
int main() {
    Box b(10);
    printWidth(b);
    return 0;
}
```



5. Function Overloading

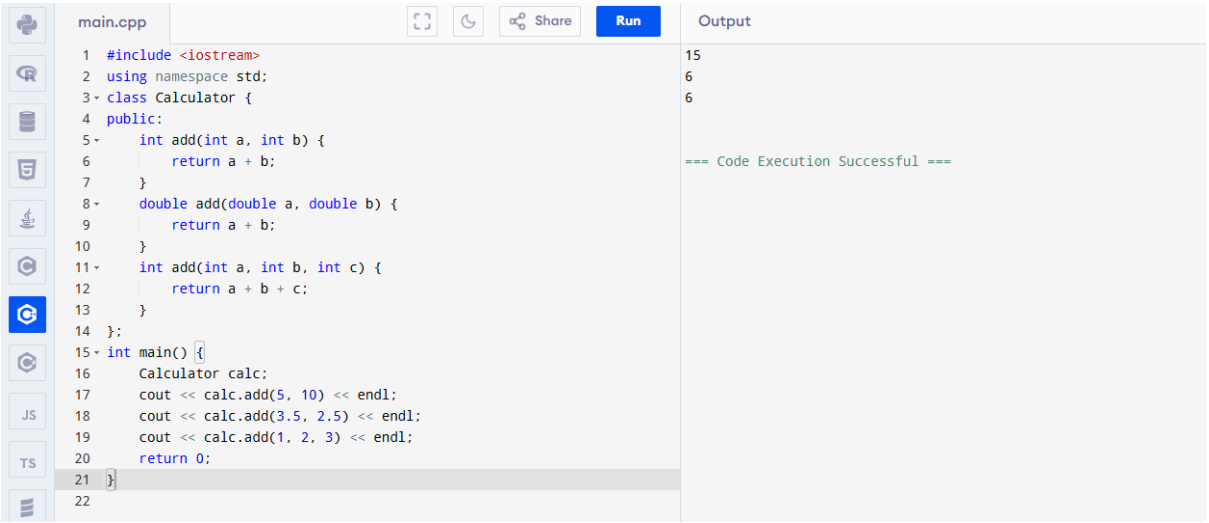
What is function overloading?

- Having **multiple functions with the same name** but different parameters (number or type).
- Allows functions to behave differently based on inputs.

Example:

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

```
class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
};

int main() {
    Calculator calc;
    cout << calc.add(5, 10) << endl;
    cout << calc.add(3.5, 2.5) << endl;
    cout << calc.add(1, 2, 3) << endl;
    return 0;
}
```



6. Operator Overloading

What is operator overloading?

- You can **give new meanings** to operators (like +, -, ==) when applied to user-defined types (classes).

Example:

#include <iostream>

using namespace std;

class Complex {

public:

int real, imag;

Complex(int r = 0, int i = 0) {

real = r;

imag = i;

}

```
    // Overload + operator
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(2, 5);
    Complex c3 = c1 + c2;  // Using overloaded +
    c3.display();
    return 0;
}
```

main.cpp

Share Run Output

```cpp
#include <iostream>
using namespace std;
class Complex {
public:
    int real, imag;
    Complex(int r = 0, int i = 0) {
        real = r;
        imag = i;
    }
    // Overload + operator
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};
int main() {
    Complex c1(3, 4), c2(2, 5);
```

```
5 + 9i


=== Code Execution Successful ===
```

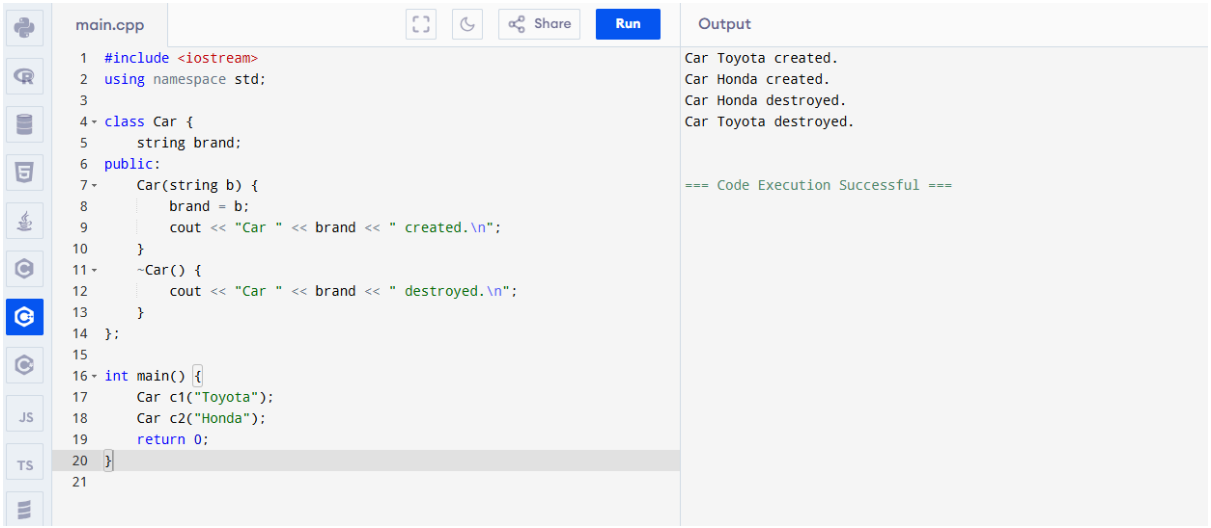
```cpp
};
int main() {
    Complex c1(3, 4), c2(2, 5);
    Complex c3 = c1 + c2;  // Using overloaded +
    c3.display();
    return 0;
}
```

Concept	Purpose	Example Use-Case
Constructor	Initialize object on creation	Set initial name, age, etc.
Destructor	Cleanup when object destroyed	Close files, free memory
this pointer	Access current object	Resolve naming conflicts
Static Members	Shared across all objects	Count how many objects created
Friend Function	Access private data from outside class	Helper functions needing special access
Function Overloading	Same function name, different params	Different ways to add numbers
Operator Overloading	Custom meaning to operators on classes	Add two complex numbers

1. Constructor & Destructor — Practice

Program: Create a class `Car` with a constructor that sets the car's brand and a destructor that says goodbye.

```
#include <iostream>

using namespace std;


class Car {

    string brand;

public:

    Car(string b) {

        brand = b;

        cout << "Car " << brand << " created.\n";

    }

    ~Car() {

        cout << "Car " << brand << " destroyed.\n";

    }

};


int main() {

    Car c1("Toyota");

    Car c2("Honda");

    return 0;

}
```



2. this Pointer — Practice

Program: Create class `Rectangle` with members `length` and `width`. Use `this` pointer in setter functions.

```
#include <iostream>

using namespace std;


class Rectangle {

    int length, width;

public:

    void setLength(int length) {

        this->length = length;

    }

    void setWidth(int width) {

        this->width = width;

    }

    int area() {

        return length * width;
```

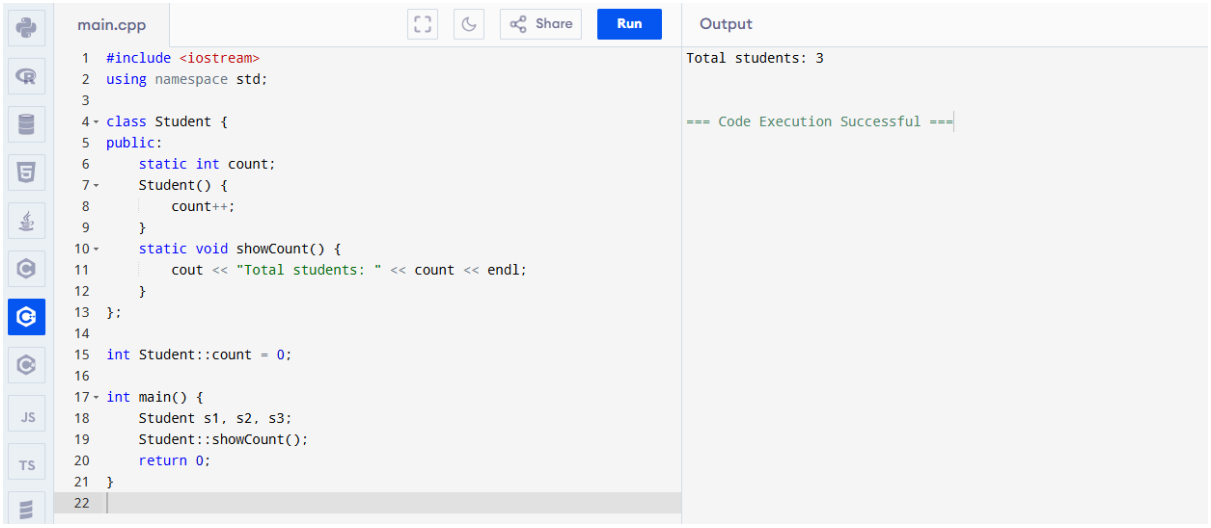
```
    }
};

int main() {
    Rectangle r;
    r.setLength(10);
    r.setWidth(5);
    cout << "Area: " << r.area() << endl;
    return 0;
}
```

3. Static Members — Practice

Program: Create class `Student`. Every time a new student is created, increase the count and show total students.

```
#include <iostream>

using namespace std;
```

```
class Student {
public:
    static int count;
    Student() {
        count++;
    }
    static void showCount() {
        cout << "Total students: " << count << endl;
    }
};

int Student::count = 0;

int main() {
    Student s1, s2, s3;
    Student::showCount();
    return 0;
}
```

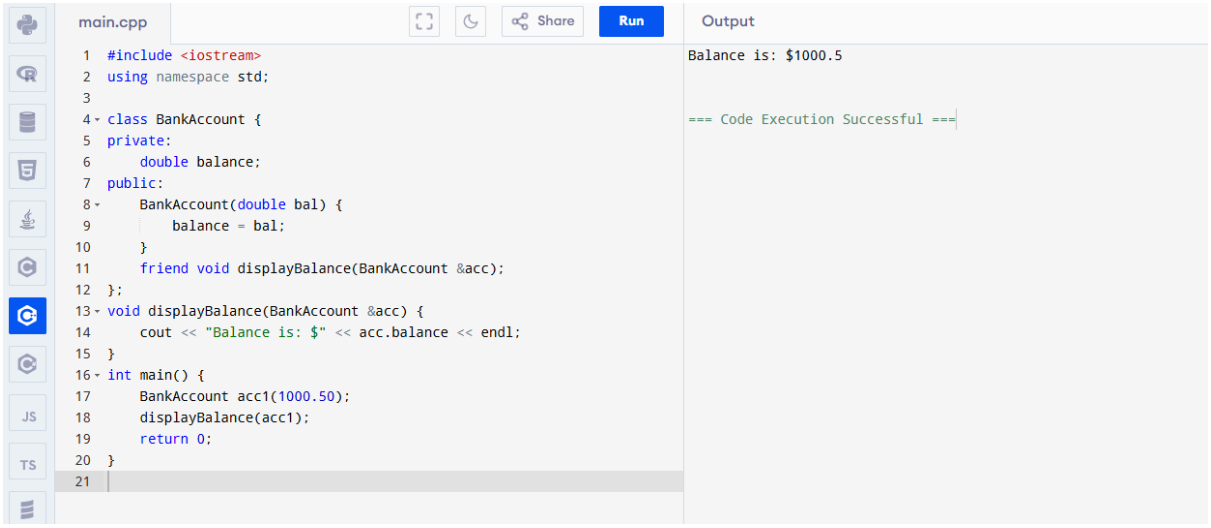


4. Friend Function — Practice

Program: Create class `BankAccount` with private `balance`. Write a friend function to display balance.

```
#include <iostream>

using namespace std;


class BankAccount {

private:

    double balance;

public:

    BankAccount(double bal) {

        balance = bal;

    }

    friend void displayBalance(BankAccount &acc);

};
```

```
void displayBalance(BankAccount &acc) {
    cout << "Balance is: $" << acc.balance << endl;
}

int main() {
    BankAccount acc1(1000.50);
    displayBalance(acc1);
    return 0;
}
```



5. Function Overloading — Practice

Program: Create a class `Printer` with overloaded functions to print `int`, `float`, and `string`.

#include <iostream>

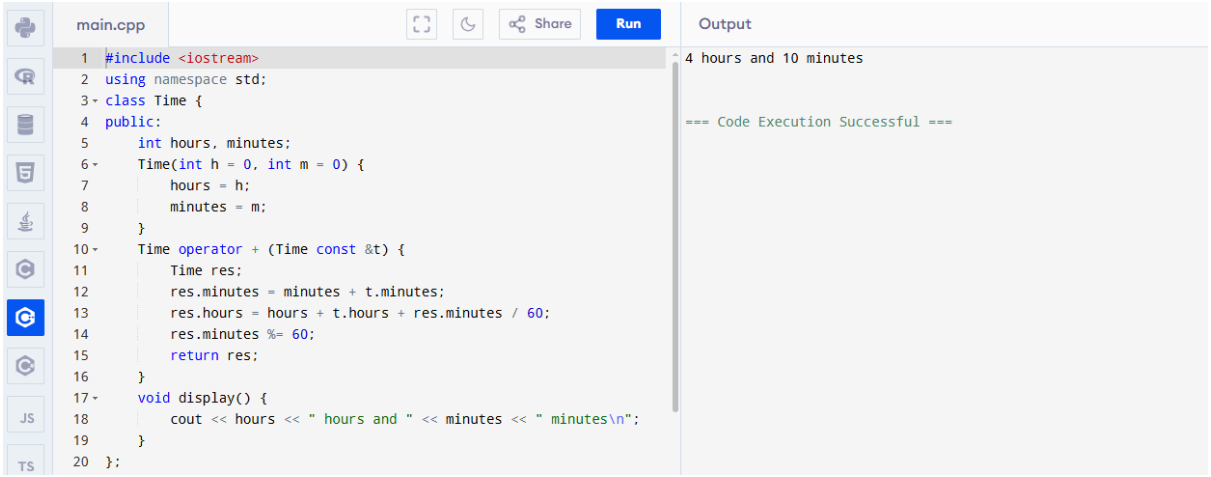
```
using namespace std;

class Printer {
public:
    void print(int i) {
        cout << "Printing int: " << i << endl;
    }
    void print(float f) {
        cout << "Printing float: " << f << endl;
    }
    void print(string s) {
        cout << "Printing string: " << s << endl;
    }
};

int main() {
    Printer p;
    p.print(5);
    p.print(3.14f);
    p.print("Hello OOP");
    return 0;
}
```

6. Operator Overloading — Practice

Program: Create class `Time` with hours and minutes, overload `+` to add two `Time` objects.

```
#include <iostream>

using namespace std;


class Time {

public:

    int hours, minutes;

    Time(int h = 0, int m = 0) {

        hours = h;

        minutes = m;

    }


    Time operator + (Time const &t) {

        Time res;

        res.minutes = minutes + t.minutes;
```

```
        res.hours = hours + t.hours + res.minutes / 60;
        res.minutes %= 60;
        return res;
    }

    void display() {
        cout << hours << " hours and " << minutes << " minutes\n";
    }
};

int main() {
    Time t1(2, 50);
    Time t2(1, 20);
    Time t3 = t1 + t2;
    t3.display();
    return 0;
}
```



```
int main() {
    Time t1(2, 50);
    Time t2(1, 20);
    Time t3 = t1 + t2;
    t3.display();
    return 0;
}

```