

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;
}

```

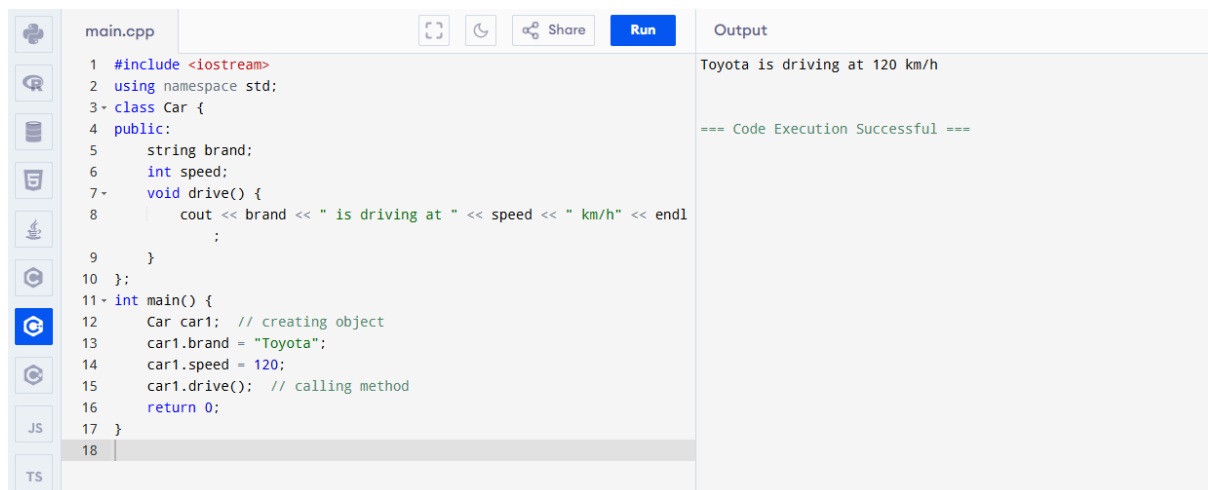
In this Program:

class Car is a blueprint.

car1 is an object of that class.

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

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



The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines a `Car` class with two data members, `brand` (string) and `speed` (int), and a member function `drive()` that prints the car's details. In the `main` function, a `Car` object named `car1` is created, its `brand` is set to "Toyota" and its `speed` to 120, and then `car1.drive()` is called. The output window on the right shows the result: "Toyota is driving at 120 km/h" and a success message "=== Code Execution Successful ===".

```

main.cpp
1 #include <iostream>
2 using namespace std;
3 class Car {
4 public:
5     string brand;
6     int speed;
7     void drive() {
8         cout << brand << " is driving at " << speed << " km/h" << endl;
9     }
10 };
11 int main() {
12     Car car1; // creating object
13     car1.brand = "Toyota";
14     car1.speed = 120;
15     car1.drive(); // calling method
16     return 0;
17 }
18
Output
Toyota is driving at 120 km/h

=== Code Execution Successful ===

```

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;
}
```

```
main.cpp
1 #include <iostream>
2 using namespace std;
3 // Creating a class
4 class Car {
5 public:
6     string brand;
7     string color;
8     int topSpeed;
9     void showDetails() {
10         cout << "Brand: " << brand << endl;
11         cout << "Color: " << color << endl;
12         cout << "Top Speed: " << topSpeed << " km/h" << endl;
13         cout << "-----" << endl;
14     }
15 };
16 int main() {
17     // Creating first object
18     Car car1;
19     car1.brand = "Honda";
20     car1.color = "Red";
21     car1.topSpeed = 180;
22
23     // Creating second object
24     Car car2;
25     car2.brand = "BMW";
26     car2.color = "Black";
27     car2.topSpeed = 240;
28     // Displaying car details
29     car1.showDetails();
30     car2.showDetails();
31     return 0;
32 }
```

Output

```
Brand: Honda
Color: Red
Top Speed: 180 km/h
-----
Brand: BMW
Color: Black
Top Speed: 240 km/h
-----

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



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

```

main.cpp
1 #include <iostream>
2 using namespace std;
3 class Student {
4 private:
5     int age; // this is hidden from outside
6 public:
7     void setAge(int a) {
8         if (a > 0)
9             age = a;
10        else
11            cout << "Invalid age!" << endl;
12    }
13    int getAge() {
14        return age;
15    }
16 };
17 int main() {
18     Student s1;
19     s1.setAge(20);    // setting the age using public function
20     cout << s1.getAge(); // getting the age using public function
21     return 0;
22 }

```

Output

```

20
=== Code Execution Successful ===

```


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;
}
```

```
main.cpp  [Icons]  Share  Run  Output  Clear

2 using namespace std;
3 class BankAccount {
4 private:
5     int balance; // private = cannot access directly
6 public:
7     void setBalance(int b) {
8         if (b >= 0) {
9             balance = b;
10        } else {
11            cout << "Invalid balance!" << endl;
12        }
13    }
14    int getBalance() {
15        return balance;
16    }
17 };
18 int main() {
19     BankAccount myAccount;
20     myAccount.setBalance(5000); // setting balance
21     cout << "Current Balance: "
22         << myAccount.getBalance(); // getting balance
23     return 0;
}
```

Current Balance: 5000

=== Code Execution Successful ===

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

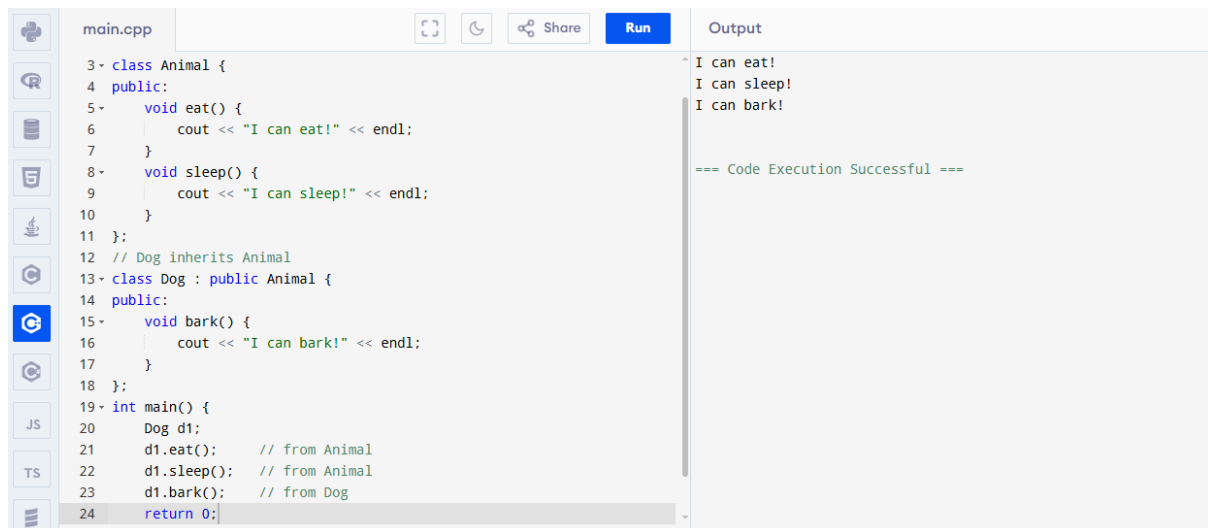
    return 0;
}
```

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()**)



```
main.cpp
3~ class Animal {
4~ public:
5~     void eat() {
6~         cout << "I can eat!" << endl;
7~     }
8~     void sleep() {
9~         cout << "I can sleep!" << endl;
10~    }
11~ };
12~ // Dog inherits Animal
13~ class Dog : public Animal {
14~ public:
15~     void bark() {
16~         cout << "I can bark!" << endl;
17~     }
18~ };
19~ int main() {
20~     Dog d1;
21~     d1.eat(); // from Animal
22~     d1.sleep(); // from Animal
23~     d1.bark(); // from Dog
24~     return 0;
}
```

Output

```
I can eat!
I can sleep!
I can bark!

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

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**

The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines a base class `Person` and a derived class `Student` that inherits from `Person`. The `main` function creates a `Student` object `s1` and calls `s1.introduce()` and `s1.showStudentInfo()`. The output window shows the results of these calls: `Name: Ali`, `Age: 20`, and `Roll Number: 101`. The code execution is successful.

```

main.cpp
1 #include <iostream>
2 using namespace std;
3 // Base class (Parent)
4 class Person {
5 public:
6     string name;
7     int age;
8     void introduce() {
9         cout << "Name: " << name << endl;
10        cout << "Age: " << age << endl;
11    }
12 };
13 // Derived class (Child)
14 class Student : public Person {
15 public:
16     int rollNo;
17     void showStudentInfo() {
18         cout << "Roll Number: " << rollNo << endl;
19     }
20 };
21 int main() {
22     Student s1;
23     // Accessing parent class members
24     s1.name = "Ali";
25     s1.age = 20;
26     // Accessing child class member
27     s1.rollNo = 101;
28     s1.introduce();    // from Person class
29     s1.showStudentInfo(); // from Student class
30     return 0;
31 }
32
Output
Name: Ali
Age: 20
Roll Number: 101

=== Code Execution Successful ===

```

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.

The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines a `Print` class with two `show` methods: one for `int` and one for `string`. The `main` function creates a `Print` object `p` and calls `p.show(10)` and `p.show("Hello")`. The output window shows the results: `Integer: 10` and `String: Hello`, followed by a success message.

```

main.cpp
1 #include <iostream>
2 using namespace std;
3 class Print {
4 public:
5     void show(int a) {
6         cout << "Integer: " << a << endl;
7     }
8     void show(string s) {
9         cout << "String: " << s << endl;
10    }
11 };
12 int main() {
13     Print p;
14     p.show(10);      // calls show(int)
15     p.show("Hello"); // calls show(string)
16     return 0;
17 }
18

```

Output

```

Integer: 10
String: Hello

=== Code Execution Successful ===

```

Type 2: Run-Time Polymorphism (Function Overriding)

Example:

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

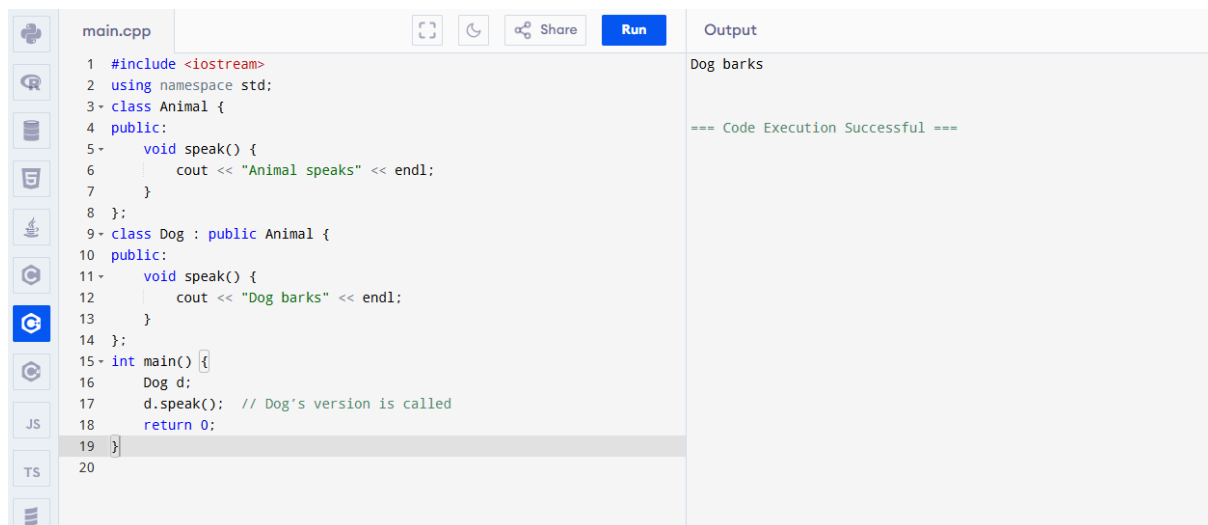
class Animal {
public:
    void speak() {
        cout << "Animal speaks" << endl;
    }
};

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.



```
main.cpp
1 #include <iostream>
2 using namespace std;
3 class Animal {
4 public:
5     void speak() {
6         cout << "Animal speaks" << endl;
7     }
8 };
9 class Dog : public Animal {
10 public:
11     void speak() {
12         cout << "Dog barks" << endl;
13     }
14 };
15 int main() {
16     Dog d;
17     d.speak(); // Dog's version is called
18     return 0;
19 }
20
```

Output

Dog barks

=== Code Execution Successful ===

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;
}
```

```
main.cpp  [Icons]  Share  Run  Output

2 using namespace std;
3 class Calculator {
4 public:
5     // Function to add two integers
6     int add(int a, int b) {
7         return a + b;
8     }
9     // Function to add two floats
10    float add(float a, float b) {
11        return a + b;
12    }
13    // Function to add three integers
14    int add(int a, int b, int c) {
15        return a + b + c;
16    }
17 };
18 int main() {
19     Calculator calc;
20     cout << "Add 2 + 3 = " << calc.add(2, 3) << endl;
21     cout << "Add 1.5 + 2.5 = " << calc.add(1.5f, 2.5f) << endl;
22     cout << "Add 4 + 5 + 6 = " << calc.add(4, 5, 6) << endl;
23     return 0;
}
```

Output

```
Add 2 + 3 = 5
Add 1.5 + 2.5 = 4
Add 4 + 5 + 6 = 15

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

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;
}

```

The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines an `Animal` base class with a `speak()` method that prints "Some animal sound". It then defines two derived classes: `Cat` and `Dog`, both of which override the `speak()` method to print "Meow!" and "Woof!" respectively. The `main()` function creates an `Animal` pointer `a1`, assigns it the address of a `Cat` object `c`, calls `a1->speak()` (outputting "Meow!"), assigns it the address of a `Dog` object `d`, calls `a1->speak()` (outputting "Woof!"), and finally returns 0. The output window on the right shows the execution results: "Meow!", "Woof!", and a success message.

```

main.cpp
1 #include <iostream>
2 using namespace std;
3 class Animal {
4 public:
5     virtual void speak() {
6         cout << "Some animal sound" << endl;
7     }
8 };
9 class Cat : public Animal {
10 public:
11     void speak() override {
12         cout << "Meow!" << endl;
13     }
14 };
15 class Dog : public Animal {
16 public:
17     void speak() override {
18         cout << "Woof!" << endl;
19     }
20 };
21
22 int main() {
23     Animal* a1;
24     Cat c;
25     Dog d;
26     a1 = &c;
27     a1->speak(); // Output: Meow!
28     a1 = &d;
29     a1->speak(); // Output: Woof!
30     return 0;
31 }

```

Output

```

Meow!
Woof!

=== Code Execution Successful ===

```



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;

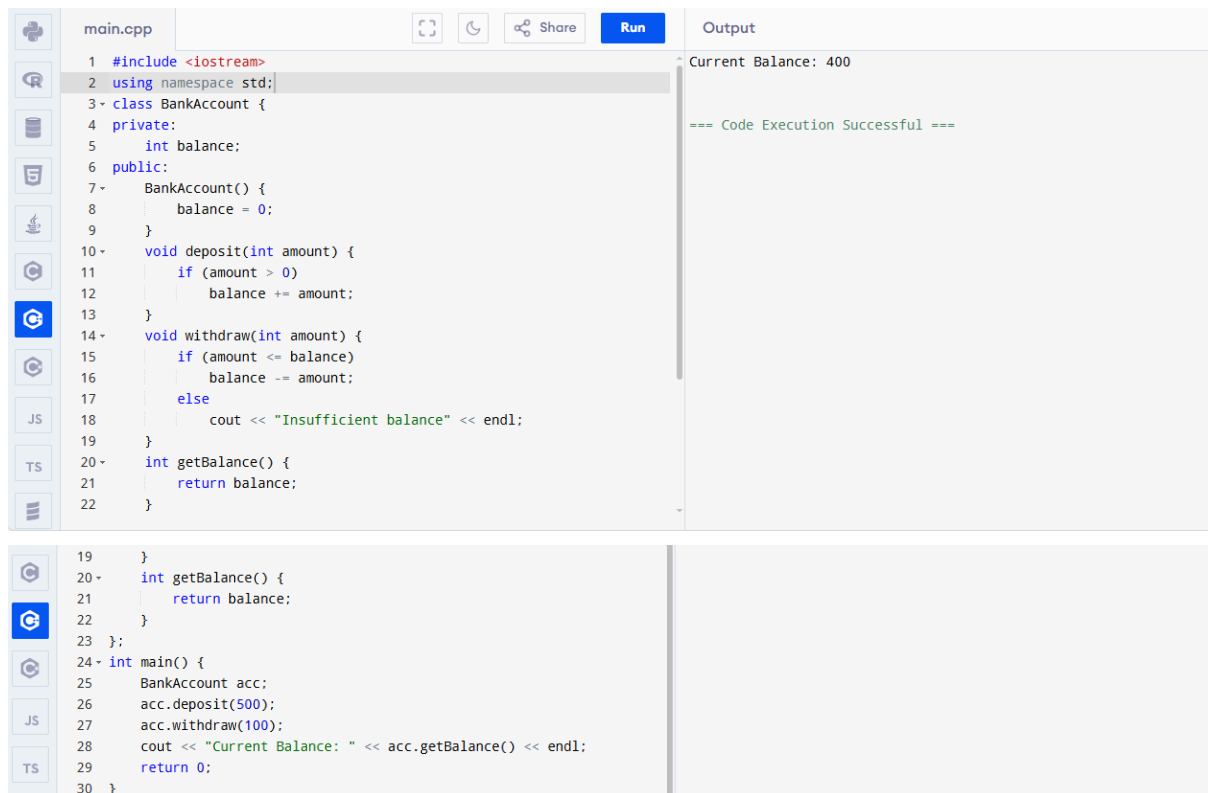
public:
    BankAccount() {
        balance = 0;
    }

    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;
}
```



```
1 #include <iostream>
2 using namespace std;
3 class BankAccount {
4 private:
5     int balance;
6 public:
7     BankAccount() {
8         balance = 0;
9     }
10    void deposit(int amount) {
11        if (amount > 0)
12            balance += amount;
13    }
14    void withdraw(int amount) {
15        if (amount <= balance)
16            balance -= amount;
17        else
18            cout << "Insufficient balance" << endl;
19    }
20    int getBalance() {
21        return balance;
22    }
23 };
24 int main() {
25     BankAccount acc;
26     acc.deposit(500);
27     acc.withdraw(100);
28     cout << "Current Balance: " << acc.getBalance() << endl;
29     return 0;
30 }
```

Output

Current Balance: 400

=== Code Execution Successful ===

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;

}

```

The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines a `Person` class with a `name` attribute, a constructor, and a destructor. The `main` function creates two `Person` objects, `p1` and `p2`, and returns 0. The output window shows the execution results: "Constructor called for Usama", "Constructor called for Ali", "Destructor called for Ali", and "Destructor called for Usama". A message at the bottom indicates "Code Execution Successful ===".

```

main.cpp
1 #include <iostream>
2 using namespace std;
3 class Person {
4 public:
5     string name;
6     // Constructor
7     Person(string n) {
8         name = n;
9         cout << "Constructor called for " << name << endl;
10    }
11    // Destructor
12    ~Person() {
13        cout << "Destructor called for " << name << endl;
14    }
15 };
16 int main() {
17     Person p1("Usama");
18     Person p2("Ali");
19     return 0;
20 }
21

```

Output

```

Constructor called for Usama
Constructor called for Ali
Destructor called for Ali
Destructor called for Usama

=== Code Execution Successful ===

```

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() {
```

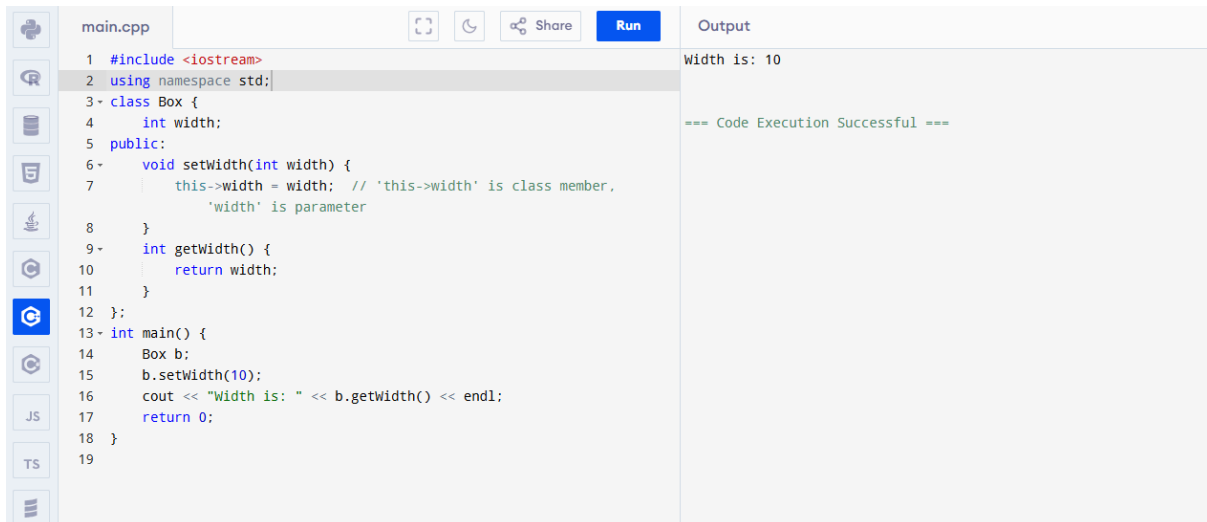
```
    Box b;
```

```
    b.setWidth(10);
```

```
    cout << "Width is: " << b.getWidth() << endl;
```

```
    return 0;
```

```
}
```



```
main.cpp
1 #include <iostream>
2 using namespace std;
3 class Box {
4     int width;
5 public:
6     void setWidth(int width) {
7         this->width = width; // 'this->width' is class member,
            'width' is parameter
8     }
9     int getWidth() {
10         return width;
11     }
12 };
13 int main() {
14     Box b;
15     b.setWidth(10);
16     cout << "Width is: " << b.getWidth() << endl;
17     return 0;
18 }
19
```

Width is: 10

=== Code Execution Successful ===

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() {

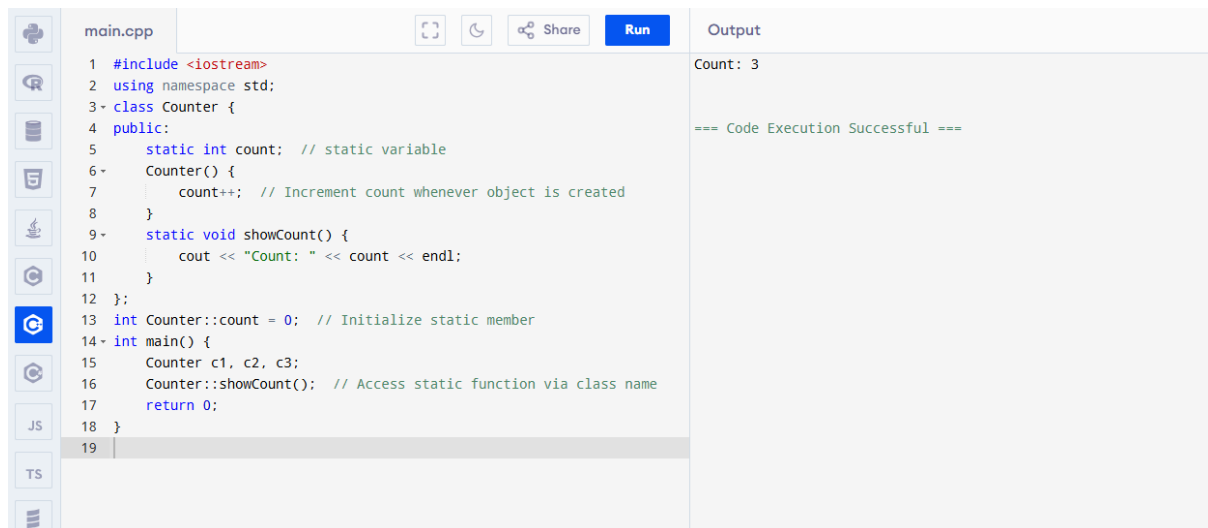
    Counter c1, c2, c3;

    Counter::showCount(); // Access static function via class name

    return 0;

}

```



The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines a `Counter` class with a static variable `count` and a static function `showCount()`. The `main` function creates three `Counter` objects and calls `Counter::showCount()`. The output window shows the result of the execution.

```

main.cpp
1 #include <iostream>
2 using namespace std;
3 class Counter {
4 public:
5     static int count; // static variable
6     Counter() {
7         count++; // Increment count whenever object is created
8     }
9     static void showCount() {
10         cout << "Count: " << count << endl;
11     }
12 };
13 int Counter::count = 0; // Initialize static member
14 int main() {
15     Counter c1, c2, c3;
16     Counter::showCount(); // Access static function via class name
17     return 0;
18 }
19

```

Output

```

Count: 3

=== Code Execution Successful ===

```

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;

}

```

The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines a `Box` class with a private member `width` and a public constructor `Box(int w)`. A friend function `printWidth` is declared and defined to access the private member. The `main` function creates a `Box` object `b` with a width of 10 and calls `printWidth(b)`. The output shows "Width is: 10" and "=== Code Execution Successful ===".

```

1 #include <iostream>
2 using namespace std;
3 class Box {
4 private:
5     int width;
6 public:
7     Box(int w) {
8         width = w;
9     }
10    // Declare friend function
11    friend void printWidth(Box box);
12 };
13 // Friend function definition
14 void printWidth(Box box) {
15     cout << "Width is: " << box.width << endl; // Access private member
16 }
17 int main() {
18     Box b(10);
19     printWidth(b);
20     return 0;
21 }

```

Output:

```

Width is: 10

=== Code Execution Successful ===

```

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;  
}
```



```
main.cpp
1 #include <iostream>
2 using namespace std;
3 class Calculator {
4 public:
5     int add(int a, int b) {
6         return a + b;
7     }
8     double add(double a, double b) {
9         return a + b;
10    }
11    int add(int a, int b, int c) {
12        return a + b + c;
13    }
14 };
15 int main() {
16     Calculator calc;
17     cout << calc.add(5, 10) << endl;
18     cout << calc.add(3.5, 2.5) << endl;
19     cout << calc.add(1, 2, 3) << endl;
20     return 0;
21 }
22
```

Output

```
15
6
6

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

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

Run

Share

```
1 #include <iostream>
2 using namespace std;
3 class Complex {
4 public:
5     int real, imag;
6     Complex(int r = 0, int i = 0) {
7         real = r;
8         imag = i;
9     }
10    // Overload + operator
11    Complex operator + (Complex const &obj) {
12        Complex res;
13        res.real = real + obj.real;
14        res.imag = imag + obj.imag;
15        return res;
16    }
17    void display() {
18        cout << real << " + " << imag << "i" << endl;
19    }
20 };
21 int main() {
22     Complex c1(3, 4), c2(2, 5);
```

Output

5 + 9i

=== Code Execution Successful ===

```
20 };
21 int main() {
22     Complex c1(3, 4), c2(2, 5);
23     Complex c3 = c1 + c2; // Using overloaded +
24     c3.display();
25     return 0;
26 }
27
```

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";
```

```
    }
```

```
};
```

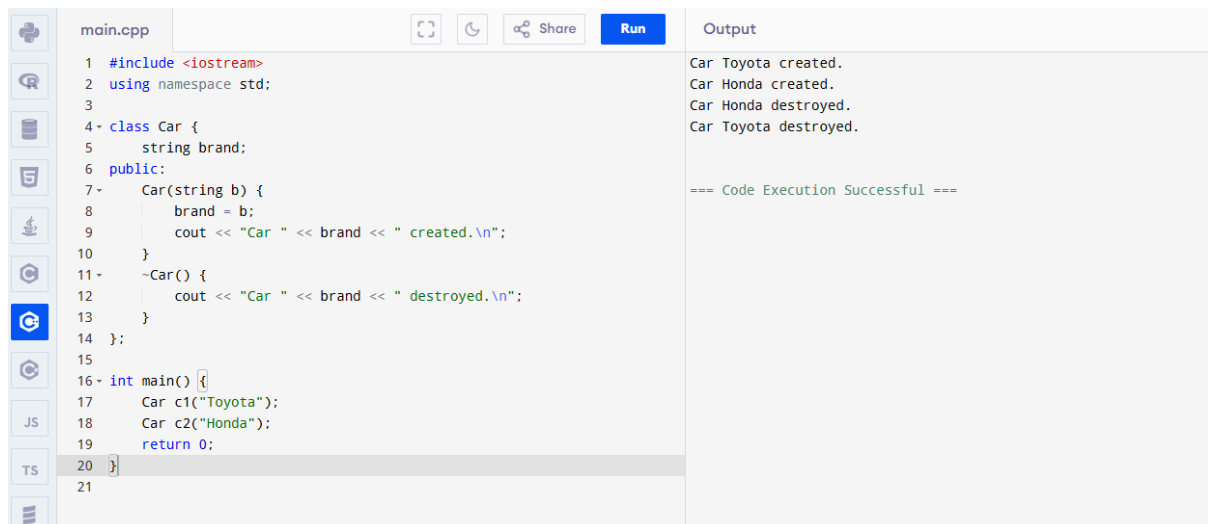
```
int main() {
```

```
    Car c1("Toyota");
```

```
    Car c2("Honda");
```

```
    return 0;
```

```
}
```



The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines a `Car` class with a `brand` string member. It has a constructor that takes a string `b` and prints "Car [brand] created.", a destructor that prints "Car [brand] destroyed.", and a `main` function that creates two cars, `c1` (Toyota) and `c2` (Honda), and returns 0. The output window on the right shows the execution results: "Car Toyota created.", "Car Honda created.", "Car Honda destroyed.", "Car Toyota destroyed.", and a success message "=== Code Execution Successful ===".

```
1 #include <iostream>
2 using namespace std;
3
4 class Car {
5     string brand;
6 public:
7     Car(string b) {
8         brand = b;
9         cout << "Car " << brand << " created.\n";
10    }
11    ~Car() {
12        cout << "Car " << brand << " destroyed.\n";
13    }
14 };
15
16 int main() {
17     Car c1("Toyota");
18     Car c2("Honda");
19     return 0;
20 }
21
```

Output:

```
Car Toyota created.
Car Honda created.
Car Honda destroyed.
Car Toyota destroyed.

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

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;

}

```

```

main.cpp
1  #include <iostream>
2  using namespace std;
3  class Rectangle {
4      int length, width;
5  public:
6      void setLength(int length) {
7          this->length = length;
8      }
9      void setWidth(int width) {
10         this->width = width;
11     }
12     int area() {
13         return length * width;
14     }
15 };
16 int main() {
17     Rectangle r;
18     r.setLength(10);
19     r.setWidth(5);
20     cout << "Area: " << r.area() << endl;
21     return 0;
22 }

```

Output

```

Area: 50

=== Code Execution Successful ===

```

3. Static Members — Practice

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

```

#include <iostream>

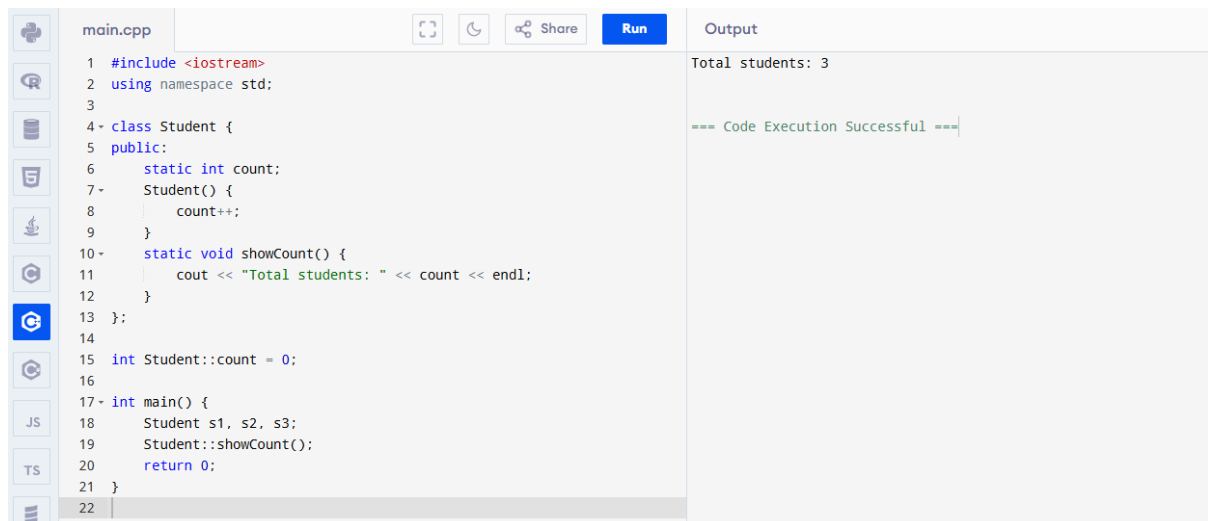
using namespace std;

```

```
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;  
}
```

```
1 #include <iostream>
2 using namespace std;
3
4 class Student {
5 public:
6     static int count;
7     Student() {
8         count++;
9     }
10    static void showCount() {
11        cout << "Total students: " << count << endl;
12    }
13 };
14
15 int Student::count = 0;
16
17 int main() {
18     Student s1, s2, s3;
19     Student::showCount();
20     return 0;
21 }
22
```

Output

Total students: 3

=== Code Execution Successful ===

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;

}
```

The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines a `BankAccount` class with a `balance` attribute and a `displayBalance` friend function. The `main` function creates a `BankAccount` object `acc1` with a balance of 1000.50 and calls `displayBalance`. The output window shows the result: "Balance is: \$1000.5" followed by a success message.

```
main.cpp
1 #include <iostream>
2 using namespace std;
3
4 class BankAccount {
5 private:
6     double balance;
7 public:
8     BankAccount(double bal) {
9         balance = bal;
10    }
11    friend void displayBalance(BankAccount &acc);
12 };
13 void displayBalance(BankAccount &acc) {
14     cout << "Balance is: $" << acc.balance << endl;
15 }
16 int main() {
17     BankAccount acc1(1000.50);
18     displayBalance(acc1);
19     return 0;
20 }
21
```

Output

```
Balance is: $1000.5

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

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;
```

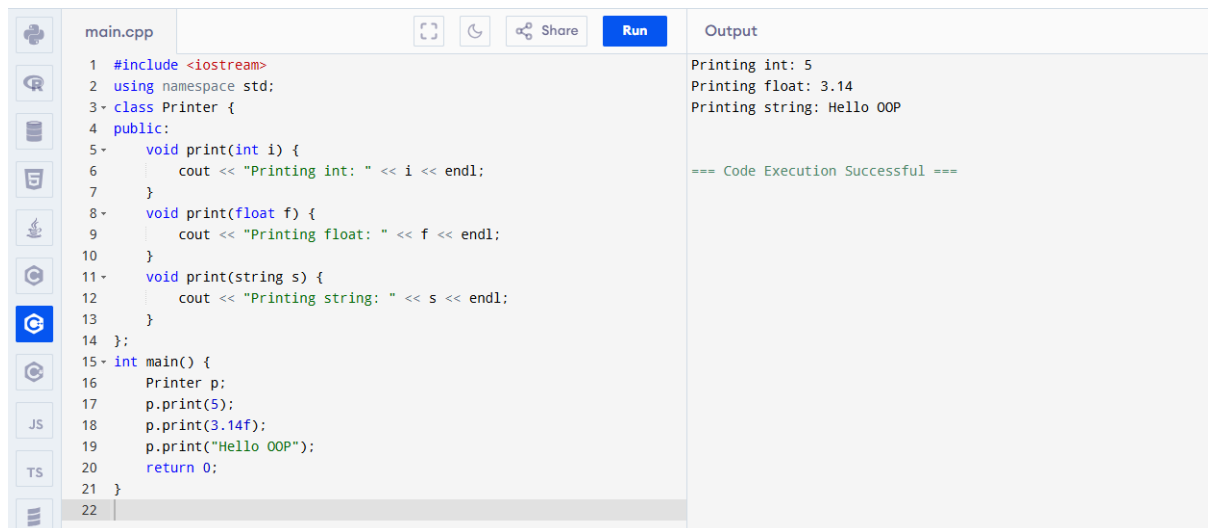
```
    p.print(5);
```

```
    p.print(3.14f);
```

```
    p.print("Hello OOP");
```

```
    return 0;
```

```
}
```



```
main.cpp
1 #include <iostream>
2 using namespace std;
3 class Printer {
4 public:
5     void print(int i) {
6         cout << "Printing int: " << i << endl;
7     }
8     void print(float f) {
9         cout << "Printing float: " << f << endl;
10    }
11    void print(string s) {
12        cout << "Printing string: " << s << endl;
13    }
14 };
15 int main() {
16     Printer p;
17     p.print(5);
18     p.print(3.14f);
19     p.print("Hello OOP");
20     return 0;
21 }
22
```

Output

```
Printing int: 5
Printing float: 3.14
Printing string: Hello OOP

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

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;

}

```

The screenshot shows a C++ IDE with a file named `main.cpp`. The code defines a `Time` class with `hours` and `minutes` attributes. It includes an addition operator `operator +` that calculates the sum of two `Time` objects, adjusting for minutes overflow. A `display` method prints the time in "hours and minutes" format. The `main` function creates `t1(2, 50)` and `t2(1, 20)`, adds them to get `t3`, and calls `t3.display()`. The output window shows "4 hours and 10 minutes" and "=== Code Execution Successful ===".

```

main.cpp
1 #include <iostream>
2 using namespace std;
3 class Time {
4 public:
5     int hours, minutes;
6     Time(int h = 0, int m = 0) {
7         hours = h;
8         minutes = m;
9     }
10    Time operator + (Time const &t) {
11        Time res;
12        res.minutes = minutes + t.minutes;
13        res.hours = hours + t.hours + res.minutes / 60;
14        res.minutes %= 60;
15        return res;
16    }
17    void display() {
18        cout << hours << " hours and " << minutes << " minutes\n";
19    }
20 };

```


Output

```

4 hours and 10 minutes

=== Code Execution Successful ===

```



```
21 • int main() {
22     Time t1(2, 50);
23     Time t2(1, 20);
24     Time t3 = t1 + t2;
25     t3.display();
26     return 0;
27 }
28
```

