Contents

Class and Objects

```cpp
#include <iostream> // Include library for input-output
using namespace std;

// Define a class
class Car {
public: // Access specifier, making the following members accessible outside
the class
    string brand;  // Field (Property) to store the brand of the car
    string model;  // Field to store the model of the car
    int year;      // Field to store the manufacturing year

    // Method (Function) to display car details
    void displayDetails() {
        cout << "Car Brand: " << brand << endl;
        cout << "Car Model: " << model << endl;
        cout << "Manufacturing Year: " << year << endl;
    }

    // Method to start the car
    void start() {
        cout << brand << " " << model << " is starting..." << endl;
    }
};

int main() {
    // Create an object (instance) of the Car class
    Car car1;

    // Assign values to the fields of car1
    car1.brand = "Toyota";
    car1.model = "Corolla";
```

```
    car1.year = 2020;

    // Call the methods on car1
    cout << "Details of Car 1:" << endl;
    car1.displayDetails(); // Display car details
    car1.start();          // Start the car

    cout << endl; // Add a blank line for separation

    // Create another object of the Car class
    Car car2;

    // Assign values to the fields of car2
    car2.brand = "Honda";
    car2.model = "Civic";
    car2.year = 2021;

    // Call the methods on car2
    cout << "Details of Car 2:" << endl;
    car2.displayDetails(); // Display car details
    car2.start();          // Start the car

    return 0;
}
Output
yaml
Copy code
Details of Car 1:
Car Brand: Toyota
Car Model: Corolla
Manufacturing Year: 2020
Toyota Corolla is starting...

Details of Car 2:
Car Brand: Honda
Car Model: Civic
Manufacturing Year: 2021
Honda Civic is starting...
```

Access Specifiers

#include <iostream>

using namespace std;

// Define a class with different access modifiers

```
class Employee {
private: // Members declared under `private` are not accessible outside the class
    int employeeID; // A private member to store employee ID

protected: // Members declared under `protected` are accessible only to derived classes
    string department; // A protected member to store department name

public: // Members declared under `public` are accessible everywhere
    string name; // A public member to store employee name

    // Public method to set the value of the private member `employeeID`
    void setEmployeeID(int id) {
        employeeID = id;
    }

    // Public method to get the value of the private member `employeeID`
    int getEmployeeID() {
        return employeeID;
    }

    // Public method to set the protected member `department`
    void setDepartment(string dept) {
        department = dept;
    }

    // Public method to display employee details
    void displayDetails() {
        cout << "Name: " << name << endl;
        cout << "Employee ID: " << employeeID << endl;
        cout << "Department: " << department << endl;
    }
```

```
};

// A derived class from Employee
class Manager : public Employee {
public:
    // Method to display department, showing access to the `protected` member
    void displayDepartment() {
        cout << "Manager Department: " << department << endl;
    }
};

int main() {
    // Create an object of the Employee class
    Employee emp;
    emp.name = "John Doe"; // Accessing the public member
    emp.setEmployeeID(101); // Setting the private member using a public method
    emp.setDepartment("HR"); // Setting the protected member using a public method
    cout << "Employee Details:" << endl;
    emp.displayDetails(); // Displaying employee details

    cout << endl;

    // Create an object of the Manager class (derived from Employee)
    Manager mgr;
    mgr.name = "Alice Smith"; // Accessing the public member
    mgr.setEmployeeID(102); // Setting the private member using a public method
    mgr.setDepartment("Finance"); // Setting the protected member using a public method
    cout << "Manager Details:" << endl;
    mgr.displayDetails(); // Displaying manager details
    mgr.displayDepartment(); // Accessing protected member through derived class method
```

```
    return 0;
}
```

Output

yaml

Copy code

```
Employee Details:
Name: John Doe
Employee ID: 101
Department: HR

Manager Details:
Name: Alice Smith
Employee ID: 102
Department: Finance
Manager Department: Finance
```

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

// Define a class with different access modifiers
class Employee {
private: // Members declared under `private` are not accessible outside the class
    int employeeID; // A private member to store employee ID

protected: // Members declared under `protected` are accessible only to derived classes
    string department; // A protected member to store department name

public: // Members declared under `public` are accessible everywhere
    string name; // A public member to store employee name

    // Public method to set the value of the private member `employeeID`
    void setEmployeeID(int id) {
        employeeID = id;
    }

    // Public method to get the value of the private member `employeeID`
    int getEmployeeID() {
        return employeeID;
    }
```

```
    // Public method to set the protected member `department`
    void setDepartment(string dept) {
        department = dept;
    }

    // Public method to display employee details
    void displayDetails() {
        cout << "Name: " << name << endl;
        cout << "Employee ID: " << employeeID << endl;
        cout << "Department: " << department << endl;
    }
};

// A derived class from Employee
class Manager : public Employee {
public:
    // Method to display department, showing access to the `protected` member
    void displayDepartment() {
        cout << "Manager Department: " << department << endl;
    }
};

int main() {
    // Create an object of the Employee class
    Employee emp;
    emp.name = "John Doe"; // Accessing the public member
    emp.setEmployeeID(101); // Setting the private member using a public
method
    emp.setDepartment("HR"); // Setting the protected member using a public
method
    cout << "Employee Details:" << endl;
    emp.displayDetails(); // Displaying employee details

    cout << endl;

    // Create an object of the Manager class (derived from Employee)
    Manager mgr;
    mgr.name = "Alice Smith"; // Accessing the public member
    mgr.setEmployeeID(102); // Setting the private member using a public
method
    mgr.setDepartment("Finance"); // Setting the protected member using a
public method
    cout << "Manager Details:" << endl;
    mgr.displayDetails(); // Displaying manager details
    mgr.displayDepartment(); // Accessing protected member through derived
class method
```

```
    return 0;
}
Output
yaml
Copy code
Employee Details:
Name: John Doe
Employee ID: 101
Department: HR

Manager Details:
Name: Alice Smith
Employee ID: 102
Department: Finance
Manager Department: Finance
```

Explanation

1. **Private Members**:
    - employeeID is private and cannot be accessed directly outside the class.
    - Public methods (setEmployeeID() and getEmployeeID()) are provided to manipulate and retrieve its value.
2. **Protected Members**:
    - department is protected and cannot be accessed directly outside the class.
    - However, it is accessible within derived classes like Manager.
3. **Public Members**:
    - name is public and can be accessed or modified directly from outside the class.
4. **Inheritance**:
    - The Manager class inherits from the Employee class.
    - The Manager class accesses the protected member department and displays it using its own method.
5. **Encapsulation**:
    - Private members are encapsulated and can only be accessed via public methods.

Constructor

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

```cpp
// Define a class
class Car {
public:
    string brand;  // Brand of the car
    int year;      // Year of manufacture

    // Constructor (automatically called when an object is created)
    Car(string b, int y) {
        brand = b; // Initialize brand
        year = y;  // Initialize year
        cout << "Constructor called! Object created.\n";
    }

    // Method to display car details
    void displayDetails() {
        cout << "Car Brand: " << brand << ", Year: " << year << endl;
    }
};

int main() {
    // Create an object of the Car class
    Car car1("Toyota", 2020); // Constructor initializes the brand and year
    car1.displayDetails();    // Display details of car1

    cout << endl; // Separator

    // Create another object of the Car class
    Car car2("Honda", 2022); // Constructor initializes the brand and year
    car2.displayDetails();    // Display details of car2

    return 0;
}
Output

Constructor called! Object created.
Car Brand: Toyota, Year: 2020

Constructor called! Object created.
Car Brand: Honda, Year: 2022
```

Explanation

1. **Constructor Definition**:
    - The Car class contains a constructor Car(string b, int y) that is called automatically when an object is created.
    - It initializes the brand and year members with the values passed as arguments.
2. **Object Creation**:

- When Car car1("Toyota", 2020) is executed, the constructor initializes the object car1 with brand = "Toyota" and year = 2020.

3. **Method Invocation**:
   - The displayDetails() method is called to display the values of the object's properties.
4. **Constructor Message**:
   - A message "Constructor called! Object created." is printed every time the constructor is invoked, showing that the constructor is automatically executed during object creation.

Copy Constructor

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

class Student {
public:
    string name;
    int age;

    // Parameterized Constructor
    Student(string n, int a) {
        name = n;
        age = a;
    }

    // Copy Constructor
    Student(const Student &s) {
        name = s.name; // Copy the name
        age = s.age;   // Copy the age
    }

    // Method to display student details
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    // Create an object using the parameterized constructor
    Student student1("John", 21);

    // Create another object using the copy constructor
    Student student2 = student1; // Copy constructor is called here

    // Display details of both objects
```

```
    cout << "Details of Student 1:" << endl;
    student1.display();

    cout << "Details of Student 2 (Copied):" << endl;
    student2.display();

    return 0;
}
Output

Details of Student 1:
Name: John, Age: 21
Details of Student 2 (Copied):
Name: John, Age: 21
```

Explanation

1. **Copy Constructor Definition**:
    - Student(const Student &s) is the copy constructor. It takes a reference to an existing object of the same class (Student) and copies its data members.
2. **Object Creation**:
    - Student student2 = student1; calls the copy constructor and initializes student2 with the values of student1.
3. **Shallow Copy**:
    - The values of name and age are directly copied from student1 to student2.
4. **Default Copy Constructor**:
    - If no copy constructor is explicitly defined, the compiler generates a default one that performs a **shallow copy**.

Inheritance

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

// Base Class for Single Inheritance
class Animal {
public:
    void eat() {
        cout << "Animal can eat." << endl;
    }
};

// Derived Class (Single Inheritance)
```

```cpp
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog can bark." << endl;
    }
};

// Derived Class (Multilevel Inheritance)
class Puppy : public Dog {
public:
    void weep() {
        cout << "Puppy can weep." << endl;
    }
};

// Base Class for Multiple Inheritance
class Mammal {
public:
    void giveBirth() {
        cout << "Mammals give birth to live young." << endl;
    }
};

// Another Base Class for Multiple Inheritance
class Bird {
public:
    void layEggs() {
        cout << "Birds lay eggs." << endl;
    }
};

// Derived Class (Multiple Inheritance)
class Bat : public Mammal, public Bird {
public:
    void fly() {
        cout << "Bat can fly." << endl;
    }
};

// Base Class for Hierarchical Inheritance
class Vehicle {
public:
    void drive() {
        cout << "Vehicle can drive." << endl;
    }
};

// Derived Classes for Hierarchical Inheritance
```

```cpp
class Car : public Vehicle {
public:
    void fuelType() {
        cout << "Car uses petrol or diesel." << endl;
    }
};

class Bike : public Vehicle {
public:
    void fuelType() {
        cout << "Bike uses petrol." << endl;
    }
};

int main() {
    // Single Inheritance
    cout << "Single Inheritance Example:" << endl;
    Dog dog;
    dog.eat();  // Inherited from Animal
    dog.bark(); // Method of Dog

    cout << "\nMultilevel Inheritance Example:" << endl;
    Puppy puppy;
    puppy.eat();   // Inherited from Animal
    puppy.bark();  // Inherited from Dog
    puppy.weep();  // Method of Puppy

    cout << "\nMultiple Inheritance Example:" << endl;
    Bat bat;
    bat.giveBirth(); // Inherited from Mammal
    bat.layEggs();   // Inherited from Bird
    bat.fly();       // Method of Bat

    cout << "\nHierarchical Inheritance Example:" << endl;
    Car car;
    car.drive();     // Inherited from Vehicle
    car.fuelType();  // Method of Car

    Bike bike;
    bike.drive();     // Inherited from Vehicle
    bike.fuelType();  // Method of Bike

    return 0;
}
```

Output

```
Single Inheritance Example:
Animal can eat.
Dog can bark.
```

```
Multilevel Inheritance Example:
Animal can eat.
Dog can bark.
Puppy can weep.

Multiple Inheritance Example:
Mammals give birth to live young.
Birds lay eggs.
Bat can fly.

Hierarchical Inheritance Example:
Vehicle can drive.
Car uses petrol or diesel.
Vehicle can drive.
Bike uses petrol.
Explanation
Single Inheritance:
```

Explanation

1. **Single Inheritance**:
    - Class Dog inherits from Animal. Methods of the Animal class (eat) are accessible in the Dog class.
2. **Multilevel Inheritance**:
    - Class Puppy inherits from Dog, and Dog inherits from Animal. Thus, Puppy inherits methods from both Dog and Animal.
3. **Multiple Inheritance**:
    - Class Bat inherits from both Mammal and Bird. It can access methods from both parent classes.
4. **Hierarchical Inheritance**:
    - Classes Car and Bike inherit from a common base class Vehicle, sharing its properties and methods.

Polymorphism

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

// Base Class
class Shape {
public:
```

```
    // Virtual method for runtime polymorphism
    virtual void draw() {
        cout << "Drawing a generic shape." << endl;
    }

    // Overloaded method for compile-time polymorphism
    void draw(int sides) {
        cout << "Drawing a shape with " << sides << " sides." << endl;
    }

    virtual ~Shape() {} // Virtual destructor
};

// Derived Class: Circle
class Circle : public Shape {
public:
    void draw() override { // Override for runtime polymorphism
        cout << "Drawing a circle." << endl;
    }
};

// Derived Class: Rectangle
class Rectangle : public Shape {
public:
    void draw() override { // Override for runtime polymorphism
        cout << "Drawing a rectangle." << endl;
    }
};

// Derived Class: Triangle
class Triangle : public Shape {
public:
    void draw() override { // Override for runtime polymorphism
        cout << "Drawing a triangle." << endl;
    }
};

int main() {
    // Compile-Time Polymorphism: Function Overloading
    Shape shape;
    cout << "Compile-Time Polymorphism Example:" << endl;
    shape.draw();           // Calls the generic draw method
    shape.draw(4);          // Calls the overloaded draw method

    cout << "\nRuntime Polymorphism Example:" << endl;

    // Runtime Polymorphism: Method Overriding
```

```
    Shape *shape1 = new Circle(); // Pointer to base class, points to derived
class
    Shape *shape2 = new Rectangle();
    Shape *shape3 = new Triangle();

    // Calls the overridden methods in derived classes
    shape1->draw(); // Circle's draw method
    shape2->draw(); // Rectangle's draw method
    shape3->draw(); // Triangle's draw method

    // Clean up
    delete shape1;
    delete shape2;
    delete shape3;

    return 0;
}
Output

Compile-Time Polymorphism Example:
Drawing a generic shape.
Drawing a shape with 4 sides.

Runtime Polymorphism Example:
Drawing a circle.
Drawing a rectangle.
Drawing a triangle.
```

Explanation

Compile-Time Polymorphism:

- Achieved using **function overloading** or **operator overloading**.
- In the example:
    - draw() with no parameters draws a generic shape.
    - draw(int sides) with a parameter specifies the number of sides.

Runtime Polymorphism:

- Achieved using **method overriding** and **virtual functions**.
- The draw() method in the Shape base class is declared as virtual.
- Derived classes (Circle, Rectangle, Triangle) override the draw() method.
- A base class pointer (Shape *) can point to derived class objects, and the appropriate draw() method of the derived class is called at runtime.