Modern Programming Principles & Practice

Feb-May 2025

sruthi.padathara@dorset.ie

Contents

- Operator Overloading: Definition, why it is required, advantages
- About operator keyword, rules of operator overloading
- Overloading various operators
- Constructors & Destructors: Definition, uses
- Types (Default Constructor, Parameter Constructor, Copy Constructor)
- Destructors
- Inheritance: Definition, Advantages
- Types of Inheritances (Single, Hierarchical, Multilevel, Multiple Hybrid)
- Implementing various kinds of inheritances
- Implementing various constructors in inheritance
- Implementing function overriding
- Implementing various access levels in inheritance

Operator Overloading: Definition, why it is required, advantages

Operator overloading in C++ allows us to redefine the meaning of operators (+, -, *, /, ==, etc.) for **user-defined data types** (classes and structures). This enables objects of a class to be used with operators just like built-in data types.

```
return_type operator symbol (arguments) {
    // Function body
}
```

Why is Operator Overloading Required?

- **Allows intuitive usage of operators with user-defined types**
  Example: c1 + c2 (instead of c1.add(c2))
- **Enhances code readability and maintainability**
  Example: Using == to compare two objects instead of writing a separate function
- **Improves efficiency in object manipulation**
  Example: Overloading [] operator for array-like access in classes

Advantages of Operator Overloading

Improves Code Readability – Makes working with objects similar to primitive data types.
Enhances Code Reusability – Allows operators to be used in different ways.
Custom Behavior for Operators – Defines operations for user-defined data types.
Better Object-Oriented Design – Encapsulates functionality within the class itself.

Operator Keyword & Rules of Operator Overloading in C++

In C++, the operator keyword is used to define **operator overloading functions**. This allows operators to work with **user-defined data types** like objects and structures.

```
return_type class_name::operator symbol (arguments) {
    // Function body
}
```

The `operator` keyword is followed by the operator symbol (`+`, `-`, `*`, etc.), making it a function name.

Rules of Operator Overloading

Operator overloading has some important rules and restrictions to ensure safe and meaningful behavior.

Rules for Operator Overloading:

1. **Must be a Class Member or Friend Function**
   - Operators must be overloaded as **member functions** or **friend functions**.
   - **Member function:** Works on this object.
   - **Friend function:** Takes objects as arguments.
2. **Cannot Overload Certain Operators**
   - Some operators **cannot** be overloaded, including:
     - :: (Scope Resolution)
     - . (Member Access)
     - .* (Pointer to Member)
     - sizeof (Size Determination)
     - typeid (Run-Time Type Identification)
3. **At Least One User-Defined Type Must Be Involved**
   - You **cannot overload an operator for primitive types**.
   - obj1 + obj2 → Allowed (where obj1 and obj2 are class objects). Correct
   - int a + int b → Not Allowed (int is a built-in type). Wrong

1. **Overloaded Operators Maintain Default Precedence & Associativity**
    - The precedence and associativity of operators **do not change** after overloading.
2. **Cannot Create New Operators**
    - Only existing operators can be overloaded. You **cannot** define a new symbol as an operator.
3. **Overloading Assignment (=), Subscript ([]), Function Call (()), and Arrow (->) Must Be Member Functions**
    - These operators **must** be overloaded as member functions inside the class.
4. **Unary and Binary Operators Have Different Overload Rules**
    - **Unary operators** (like -, ++, --) take **no parameters** in member functions.
    - **Binary operators** (like +, -, *, /) take **one parameter** in member functions.

Summary

operator keyword is used for overloading operators.
Rules:

- Must involve a **user-defined type**.
- Certain operators **cannot** be overloaded.
- Cannot **change precedence** of operators.
- Some operators **must be member functions** (=, (), [], ->).
  Example: Overloading + for complex numbers.

Unary Operator Overloading:
Write a C++ program to create a Counter class with an integer data member. Overload the ++ operator (both prefix and postfix) to increment the value of the counter. Display the updated value after each increment.

Binary Operator Overloading (+ Operator):
Define a Complex class that represents complex numbers. Overload the + operator to add two complex numbers and return the result as a new object.

Relational Operator Overloading (== Operator):
Implement a Point class with x and y coordinates. Overload the == operator to compare two points. If both the x and y coordinates are equal, return true; otherwise, return false.

Subscript Operator Overloading ([] Operator):
Create an Array class that holds five integer values. Overload the [] operator to allow accessing array elements based on an index value. Display a message if the index is out of bounds.

Function Call Operator Overloading (() Operator):
Write a program to overload the function call () operator in a class named Multiply. The class should accept two integers and return their product.

Arrow Operator Overloading (-> Operator):
Implement a SmartPointer class that overloads the -> operator to access the members of a Test class. Demonstrate how a smart pointer can be used to call a member function of Test.

Overloading - Operator:
Write a C++ program to overload the - operator for a Vector class that subtracts two vectors.

Overloading Stream Operators (<< and >>):
Create a Student class with attributes like name and marks. Overload the << operator for output and >>operator for input.

Overloading - Operator:
Write a C++ program to overload the - operator for a Vector class that subtracts two vectors.

Overloading Stream Operators (<< and >>):
Create a Student class with attributes like name and marks. Overload the << operator for output and >>operator for input.

Constructors & Destructors in C++

A **constructor** is a special member function in C++ that is automatically called when an object of a class is created. It is used to **initialize** objects.

Characteristics of a Constructor:

- **Same name as the class.**
- **No return type (not even void).**
- **Called automatically when an object is created.**
- **Can be overloaded (multiple constructors with different parameters).**

```
class ClassName {
public:
    ClassName() { // Constructor
        cout << "Constructor called!" << endl;
    }
};
```

Types of Constructors:

1. **Default Constructor:** A constructor that takes no parameters and initializes the object with default values.
2. **Parameterized Constructor:** A constructor that accepts arguments to initialize an object with specific values.
3. **Copy Constructor:** A constructor that initializes an object by copying another object of the same class.
4. **Move Constructor:** Used for moving resources (introduced in C++11).
5. **Constructor with Default Arguments:** A constructor where parameters have default values.

Destructor

A **destructor** is a special member function in C++ that is automatically called when an object is destroyed. It is used to **free resources** such as memory allocation, file handling, etc.

Characteristics of a Destructor:

- **Same name as the class, but preceded by ~ (tilde) symbol.**
- **No return type and no parameters.**
- **Only one destructor per class (cannot be overloaded).**
- **Called automatically when an object goes out of scope.**

```
class ClassName {
public:
    ~ClassName() { // Destructor
        cout << "Destructor called!" << endl;
    }
};
```

Uses of Constructors & Destructors

Uses of Constructors:

- Initialize objects automatically when they are created.
- Reduce code repetition by setting default values.
- Can enforce constraints (e.g., checking valid input values).

Uses of Destructors:

- Free dynamically allocated memory.
- Close files and release system resources.
- Prevent memory leaks and improve resource management.

Inheritance in C++

Definition of Inheritance

- Inheritance allows a class (child) to acquire properties and behaviors of another class (parent).
- Promotes code reusability and hierarchical classification.

Advantages of Inheritance

- Reduces code duplication.
- Enhances code maintenance.
- Promotes reusability of existing code.
- Provides a clear hierarchical structure.

Types of Inheritance

Single Inheritance – One class derives from another.
Multilevel Inheritance – A derived class inherits from another derived class.
Multiple Inheritance – A class inherits from two or more base classes.
Hierarchical Inheritance – Multiple derived classes inherit from a single base class.
Hybrid Inheritance – Combination of multiple inheritance types, often requiring virtual base classes.

Single Inheritance

- **Definition**: A class (derived class) inherits properties and behaviors from one base class only.
- **Example**: A Car class inherits from a Vehicle class.
- **Advantages**:
    - Simple structure.
    - Easier to maintain and understand.

Example:

Create a class Person with a method introduce() that prints a message introducing the person. Create a derived class Student that inherits from Person and adds a method study().

- Implement a main() function that creates a Student object and calls both introduce() and study()

Multiple Inheritance

- **Definition**: A class inherits from more than one base class.
- **Example**: A FlyingCar class inherits from both Car and Plane classes.
- **Advantages**:
    - Allows combining features of multiple base classes.
    - Increases flexibility in object-oriented designs.

Example

Create two classes Bird and Vehicle. In the Bird class, add a method fly() that prints "Flying". In the Vehicle class, add a method drive() that prints "Driving". Then, create a class FlyingCar that inherits from both Bird and Vehicle and has a method flyAndDrive() which calls both fly() and drive().

- Implement a main() function that creates a FlyingCar object and calls flyAndDrive().

Multilevel Inheritance

- **Definition**: A derived class inherits from another derived class, creating a chain of inheritance.
- **Example**: A Person class is inherited by Employee class, which is then inherited by Manager class.
- **Advantages**:
    - Represents a hierarchical relationship.
    - Allows more detailed functionality with each level.

Example:

Create three classes: Animal, Mammal, and Bird. The Animal class should have a method eat() that prints "Eating". The Mammal class should have a method walk() that prints "Walking". The Bird class should have a method chirp() that prints "Chirping". Finally, create a Bat class that inherits from both Mammal and Bird, and has its own method fly().

- Implement a main() function that creates a Bat object and calls eat(), walk(), chirp(), and fly().

Hierarchical Inheritance

- **Definition**: Multiple derived classes inherit from a single base class.
- **Example**: A Bird class is inherited by Sparrow, Eagle, and Parrot classes.
- **Advantages**:
    - Saves memory and resources.
    - Promotes code reuse across multiple derived classes.

Example

Create a class Shape with a method draw() that prints "Drawing a shape". Then, create two derived classes Circleand Rectangle that inherit from Shape. Add a specific method in each derived class: Circle should have drawCircle() and Rectangle should have drawRectangle().

- Implement a main() function that creates objects of Circle and Rectangle, and calls their respective methods.

Hybrid Inheritance

- **Definition**: A combination of more than one type of inheritance (e.g., multiple inheritance and multilevel inheritance).
- **Example**: A SuperManager class inherits from both Employee (single inheritance) and Manager (multilevel inheritance).
- **Challenges**:
    - Ambiguity in member functions or data members from multiple base classes.
    - Can be resolved using **virtual inheritance**.

Example

- **Create three classes: Animal, Mammal, and Bird. The Animal class should have a method eat() that prints "Eating". The Mammal class should have a method walk() that prints "Walking". The Bird class should have a method chirp() that prints "Chirping". Finally, create a Bat class that inherits from both Mammal and Bird, and has its own method fly().**
    - Implement a main() function that creates a Bat object and calls eat(), walk(), chirp(), and fly().

Implementing Constructors in Inheritance

Constructor Execution Order

- **Rule**: In **multilevel inheritance**, constructors of base classes are called first, followed by derived class constructors.
- **Example**:
    - Grandparent constructor → Parent constructor → Child constructor.

Default Constructor in Inheritance

- **Definition**: A constructor that doesn’t take any parameters and initializes objects with default values.
- **Usage**: Used when no arguments are passed during object creation.
- **Example**: Person() initializes default values for attributes like name and age.

Parameterized Constructor in Inheritance

- **Definition**: A constructor that accepts parameters to initialize the object with specific values.
- **Usage**: Used when user-defined values are needed for object initialization.
- **Example**: Person(string name, int age) initializes a Person object with the provided values.

Copy Constructor in Inheritance

- **Definition**: A constructor that initializes an object by copying values from another object of the same class.
- **Usage**: Ensures deep copying of objects in inheritance.
- **Example**: Person(const Person& other) copies the attributes of another Person.

Constructor with Default Arguments in Inheritance

- **Definition**: A constructor that provides default values for its parameters.
- **Usage**: Useful in scenarios where parameters are not always provided.
- **Example**: Person(string name = "Unknown", int age = 30) initializes name and age with default values if not provided.

Constructor Overloading in Inheritance

- **Definition**: Multiple constructors with different parameter lists in a class or its derived classes.
- **Usage**: Allows creating objects with different initialization logic.
- **Example**:
    - Person() – default constructor
    - Person(string name) – parameterized constructor
    - Person(string name, int age) – constructor with both parameters.

Constructor Chaining in Inheritance

Create a class Base with a parameterized constructor that takes an integer and prints it. Create a class Derived that inherits from Base and calls the Base constructor using : Base(10). In the Derived class, add a method display() that prints "Derived class method".

- Implement a main() function that creates a Derived object and calls the display() method.

Accessing Base Class Members in Derived Class

Create a base class Vehicle with a private member speed and a public method setSpeed(int s) that sets the speed. Create a derived class Car that inherits from Vehicle and has a public method displaySpeed() that prints the speed of the car.

- Implement a main() function where you create a Car object, set the speed using setSpeed(), and call displaySpeed().

Virtual Inheritance (Avoiding Diamond Problem)

Create a base class Shape with a method draw(). Create two classes Rectangle and Circle that inherit from Shape. Then, create a class Square that inherits from both Rectangle and Circle. Implement a virtual destructor in the base class to avoid the diamond problem.

- Implement a main() function that creates a Square object and calls the draw() method.

Using super() to Call Parent Class Constructor

Create a class Person with a parameterized constructor that takes name and age. Create a class Student that inherits from Person and uses super() to call the Person constructor. The Student class should also have a method display() to print the name and age.

- Implement a main() function that creates a Student object and calls display().

Multiple Constructors in Inheritance

- **Create a base class Shape with a constructor that initializes the shape's color. Create a derived class Circlewith a constructor that initializes the radius. Also, implement a constructor in Circle that calls the base class constructor and initializes the color.**
    - Implement a main() function that creates a Circle object and prints the color and radius.

Polymorphism in Inheritance

Create a base class Employee with a method calculateSalary(). Create derived classes Manager, Engineer, and Intern, each of which overrides calculateSalary() with different implementations.

- Implement a main() function where you create objects of Manager, Engineer, and Intern, and call calculateSalary() for each object. Demonstrate polymorphism by storing these objects in a base class pointer array.