Object Oriented Programming with C++

Tutorial 2

Recap : Introduction to Object-Oriented Programming

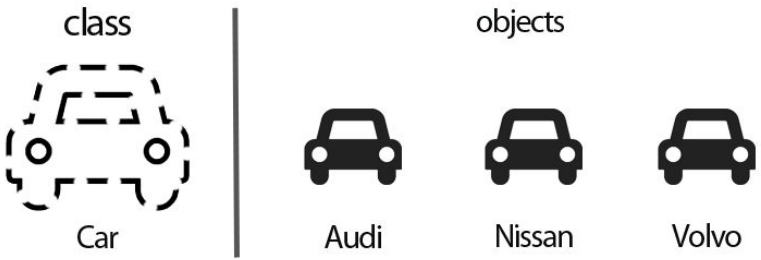
What is OOP?

Object-Oriented Programming is a programming paradigm based on the concept of "objects" that contain data and functions.

Organizes software design around objects rather than functions and logic

Enables modular, reusable, and maintainable code

Represents real-world entities as software objects

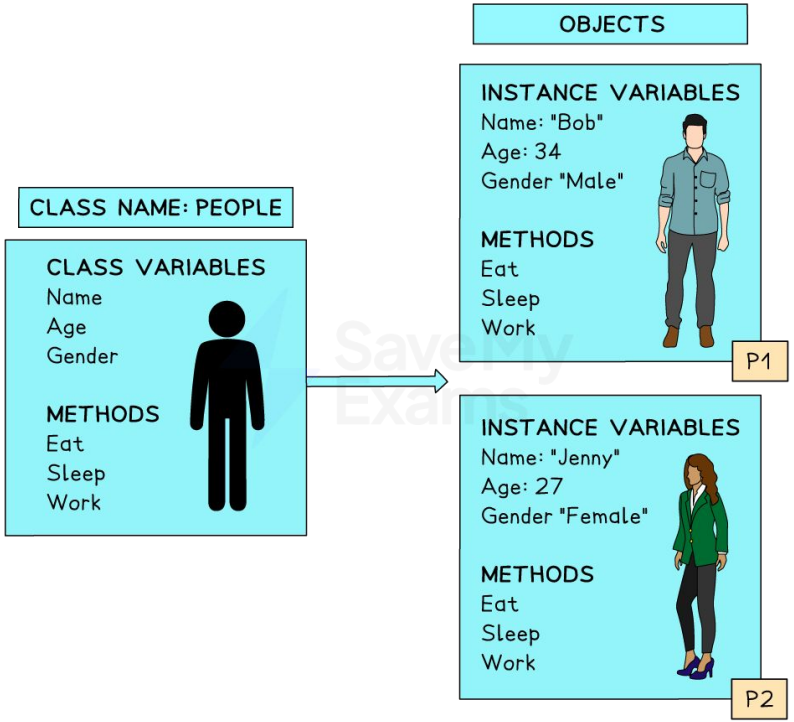


Key OOP Concepts

- **Encapsulation:** Bundling data and methods that operate on that data
- **Inheritance:** A new class of object derives properties and characteristics from another class
- **Polymorphism:** Objects can take different forms depending on context
- **Abstraction :** Hiding complex implementation details

Recap : Class and Objects

Class and Objects




```
};
```

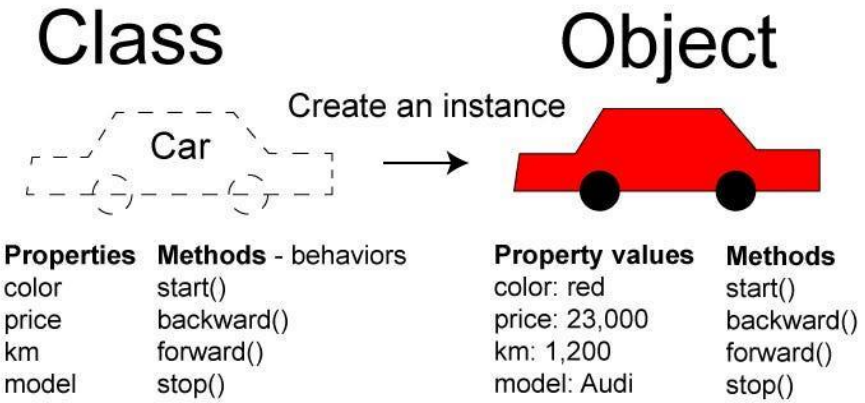
Recap : Member Functions and Data Members

Data Members (Attributes)

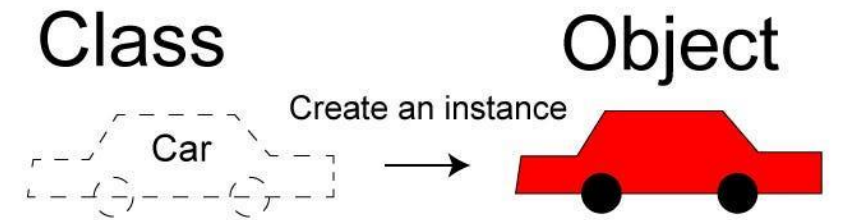
- Variables declared within a class
- Represent the state/properties of an object
- Can have different access levels (public, private, protected)

Member Functions (Methods)

- Functions declared within a class
- Define the behavior of objects
- Can access and modify data members
- Can be declared inside the class and defined outside



Recap : Example Class

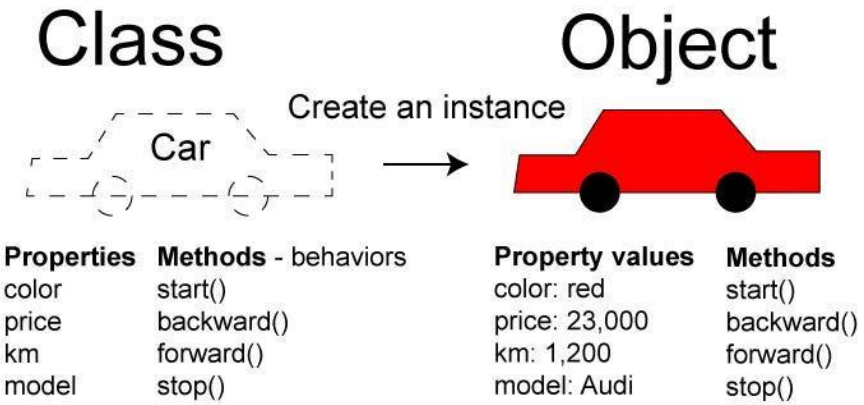


Properties	Methods - behaviors
color	start()
price	backward()
km	forward()
model	stop()

Property values	Methods
color: red	start()
price: 23,000	backward()
km: 1,200	forward()
model: Audi	stop()

```
1   #include <iostream>
2
3   using namespace std;
4
5   class Car {
6       public:
7           int price;
8           int km;
9           string color;
10          string model;
11
12          void start();
13          int forward();
14          int backward();
15          void stop();
16  };
17
```

Recap : Object Creation




```
18  int main() {
19      Car audi;
20
21      audi.price = 2000;
22      audi.km = 500;
23      audi.color = "black";
24      audi.model = "b200";
25
26      cout<<"Color:" <<audi.color<<endl;
27  }
```

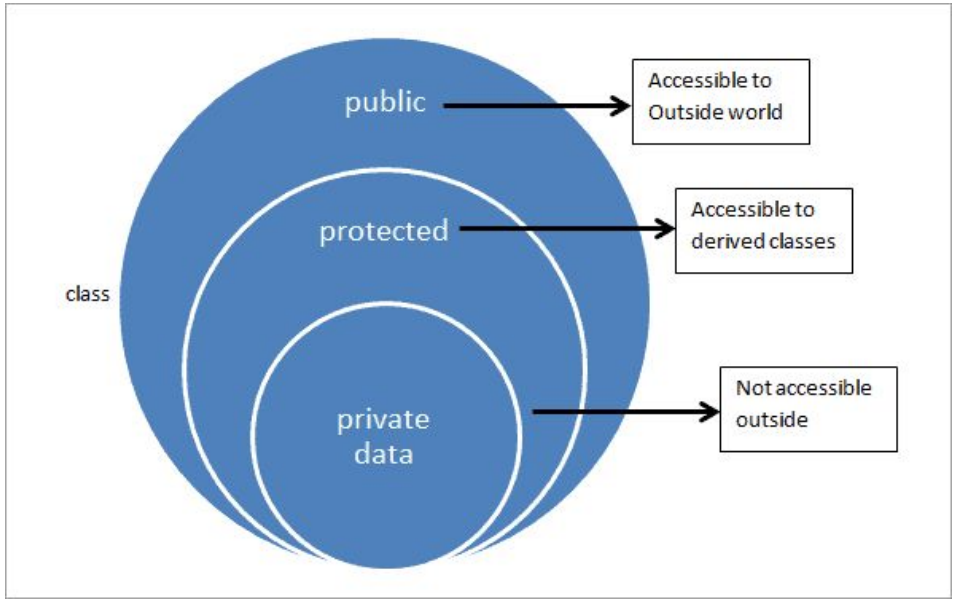
Recap : Access Specifiers

Access Specifiers in C++

Access specifiers define the accessibility of class members:

- **public:** Accessible from anywhere
- **private:** Accessible only within the class
- **protected:** Accessible within the class and derived classes

Access specifiers are fundamental to encapsulation, one of the core principles of OOP.



Best Practices

- Keep data members private
- Provide public methods to access and modify private data
- Use protected for members that derived classes need access to

Constructors and Destructors

What are Constructors ?

- Special member function automatically called when an object is created.
- Same name as the class.
- No return type (not even void).
- Can be:
  - Default Constructor – no parameters
  - Parameterized Constructor – with parameters
  - Copy Constructor – creates a copy of another object

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

class Car {
public:
    // Constructor
    Car() {
        cout << "A Car object is created!" << endl;
    }
};

int main() {
    Car myCar; // Constructor is automatically called
    return 0;
}
```

Constructors

Default Constructor

- No parameters.
- Automatically called when an object is created.
- Can be **compiler-provided** or **user-defined**.
- If no constructor is defined, C++ provides a default one.

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

class Car {
public:
    // Constructor
    Car() {
        cout << "A Car object is created!" << endl;
    }
};

int main() {
    Car myCar; // Constructor is automatically called
    return 0;
}
```

Constructors

Parameterized Constructor

- Takes parameters to initialize object with specific values.
- Allows custom initialization at creation.
- Used to pass values during object creation.

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

class Car {
public:
    string model;

    Car(string m) {
        model = m;
    }
};

int main() {
    Car car("BMW");
    cout << "Car model: " << car.model << endl;
    return 0;
}
```

Constructors

Copy Constructor

- Initializes a new object as a copy of an existing object.
- Automatically called when a copy of an object is needed.
- There are two types
    - Default Copy Constructor
    - User-Defined Copy Constructor

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

class Car {
public:
    string model;

    Car(string m) {
        model = m;
    }

    // Copy Constructor
    Car(const Car &c) {
        model = c.model;
    }
};


int main() {
    Car car1("BMW");
    cout << "Car 1 Model : " << car1.model << endl;

    Car car2 = car1;
    cout << "Car 2 Model : " << car2.model << endl;

    return 0;
}
```

Constructors

Default Copy Constructor

- Automatically provided by the compiler if you don’t define your own.
- Performs a shallow copy (copies values as-is, including pointers).

User-Defined Copy Constructor

- You define it when you need to perform a deep copy
- e.g., copying the contents of dynamically allocated memory.

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

class Person {
public:
    char* name;

    // Constructor
    Person(const char* n) {
        name = new char[strlen(n) + 1];
        strcpy(name, n);
        cout << "Constructor called.\n";
    }

    // User-defined Copy Constructor (Deep Copy)
    Person(const Person& p) {
        name = new char[strlen(p.name) + 1];
        strcpy(name, p.name);
        cout << "Copy Constructor called (Deep Copy).\n";
    }
};

int main() {
    Person p1("John");
    Person p2 = p1;  // Invokes copy constructor

    cout << "Name: " << p1.name << endl;
    cout << "Name: " << p2.name << endl;

    return 0;
}
```

Constructors

Shallow copy vs Deep copy

Feature	Shallow Copy	Deep Copy
Definition	Copies only the memory addresses (pointers).	Copies the actual data, allocates separate memory.
Memory Allocation	Shares the same memory between objects.	Allocates **new memory** for the copied data.
Side Effects	Changing one object affects the other.	Objects are independent after copying.
Use Case	Safe when no dynamic memory is used.	Required when using **dynamic memory**.

Destructors

What are Destructors ?

- Special member function automatically called when an object is destroyed.
- Same name as the class but with a tilde (~) prefix.
- No parameters, no return type.
- Used to free resources like memory or file handles.

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

class Car {
public:
    string model;

    Car(string m) {
        model = m;
    }

    ~Car() {
        cout << "Destructor Called\n";
    }
};

int main() {
    Car car("BMW");
    cout << "Car model: " << car.model << endl;
    return 0;
}
```

Constructors and Destructors - Summary

```cpp
class Car {
private:
    string brand;
    int price;

public:
    // Default Constructor
    Car() {
        cout << "Default constructor called. Brand: " << brand << endl;
    }

    // Parameterized Constructor 1
    Car(string b) {
        brand = b;
        cout << "Parameterized constructor 1 called. Brand: " << brand << endl;
    }

    // Parameterized Constructor 2
    Car(string b, int p) {
        brand = b;
        price = p;
        cout << "Parameterized constructor 2 called. Brand: " << brand << " Price: " << price << endl;
    }

    // Copy Constructor
    Car(const Car &c) {
        brand = c.brand;
        cout << "Copy constructor called. Brand copied: " << brand << endl;
    }

    // Destructor
    ~Car() {
        cout << "Destructor called for brand: " << brand << endl;
    }
};
```

```cpp
int main() {
    Car car1;               // Default constructor
    Car car2("BMW");        // Parameterized constructor 1
    Car car3("BMW", 1000);  // Parameterized constructor 2
    Car car4 = car2;        // Copy constructor

    return 0;
}
```

The static Keyword

Why static keyword ?

- Allocates memory only once during the program lifetime.
- Retains value between function calls.
- Useful when a variable or function does not depend on specific objects.

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

void counterDemo() {
    static int staticCount = 0;  // static local variable
    int normalCount = 0;         // normal local variable

    staticCount++;
    normalCount++;

    cout << "Static Count: " << staticCount
         << " | Normal Count: " << normalCount << endl;
}

int main() {
    counterDemo();
    counterDemo();
    counterDemo();
    return 0;
}
```

The static Keyword

Static Variables in Functions

- Retain values between function calls.

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

void counterDemo() {
    static int staticCount = 0;  // static local variable
    int normalCount = 0;         // normal local variable

    staticCount++;
    normalCount++;

    cout << "Static Count: " << staticCount
         << " | Normal Count: " << normalCount << endl;
}

int main() {
    counterDemo();
    counterDemo();
    counterDemo();
    return 0;
}
```

```
Static Count: 1 | Normal Count: 1
Static Count: 2 | Normal Count: 1
Static Count: 3 | Normal Count: 1

--------------------------------
Process exited after 0.0311 seconds wi
Press any key to continue . . .
```

The static Keyword

Static Variables (Data Members) in Classes

- Shared across all objects
- Not initialized via constructors.
- Must be initialized outside the class using scope resolution

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

class Car {
    public:
        int normalCount;
        static int staticCount;  // static data member

    Car(int c) {
        normalCount = c;
    }
};

// Definition of static member outside the class
int Car::staticCount = 3;

int main() {
    Car c1(1);
    Car c2(2);

    cout << "Car 1 Static Count: " << c1.staticCount
         << " | Normal Count: " << c1.normalCount << endl;

        cout << "Car 2 Static Count: " << c2.staticCount
         << " | Normal Count: " << c2.normalCount << endl;

    cout << "For any Car static count: " << Car::staticCount << endl;
}
```

```
Car 1 Static Count: 3 | Normal Count: 1
Car 2 Static Count: 3 | Normal Count: 2
For any Car static count: 3
```

The static Keyword

Static Member Functions in Classes

- Belong to class, not object.
- No this pointer, can't access non-static members.

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

class Car {
public:
    static int totalCars;
    static void showTotal() {
        cout << "Total cars: " << totalCars << endl;
    }
};

int Car::totalCars = 10;

int main() {
    Car::showTotal();
}
```

The static Keyword

Static Class Objects

- Like variables, objects also when declared as static have a scope till the lifetime of the program.
- Exists the scope at the end of the program.

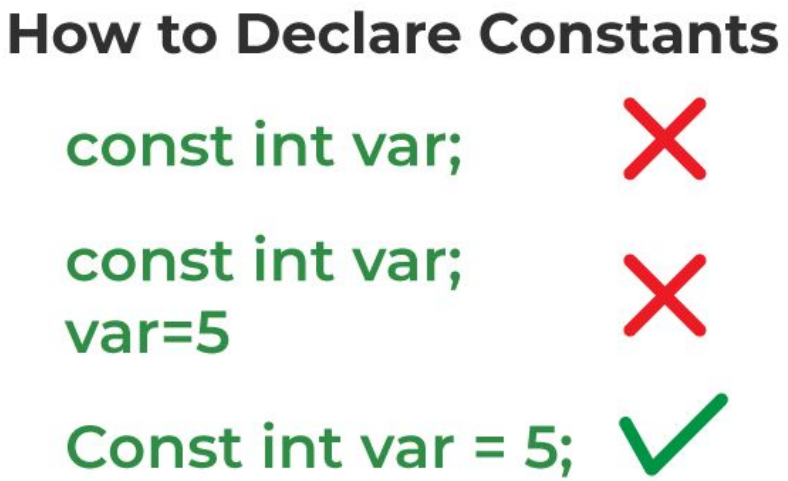
```cpp
#include <iostream>
#include <string>
using namespace std;

class Car {
private:
    string brand;
    string model;
    int year;

public:
    // Constructor
    Car(string b, string m, int y) {
        brand = b;
        model = m;
        year = y;
        cout << "Car object created\n";
    }

    // Member Function
    void displayInfo() {
        cout << "Brand: " << brand << ", Model: " << model << ", Year: " << year << endl;
    }

    // Destructor
    ~Car() {
        cout << "Car object destroyed\n";
    }
};
```

```cpp
void showStaticCar() {
    static Car myStaticCar("Honda", "Civic", 2018);
    myStaticCar.displayInfo();
}

int main() {
    cout << "First call to showStaticCar()\n";
    showStaticCar();

    cout << "\nSecond call to showStaticCar()\n";
    showStaticCar();

    cout << "\nEnd of main\n";
    return 0;
}
```

The const Keyword

What are constant variables ?

- Variables whose values cannot be changed.
- Must be initialized at the time of declaration.



The const Keyword

What are constant variables ?

- Variables whose values cannot be changed.
- Must be initialized at the time of declaration.

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

int main() {
    const int maxSpeed = 120;  // constant variable

    cout << "The maximum speed is: " << maxSpeed << " km/h" << endl;

    maxSpeed = 150;  // ? Error: assignment of read-only variable ‘maxSpeed’

    return 0;
}
```

The const Keyword

Pointers with const keyword

- `const int* ptr;`
  - A pointer to a constant integer
  - You can't change the value ptr points to,
  - But you can change the pointer itself to point to another integer.

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

int main() {
    int a = 10;
    int b = 20;

    const int* ptr = &a;  // pointer to const int

    // *ptr = 15;  // ? Not allowed: cannot modify the value
    ptr = &b;      // ? Allowed: can point to another address

    cout << "Value pointed by ptr: " << *ptr << endl;
    return 0;
}
```

- `int* const ptr;`
  - A constant pointer.
  - You can change the value at the memory location it points to,
  - But you cannot make the pointer point to a different address.

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

int main() {
    int a = 10;
    int b = 20;

    int* const ptr = &a;  // const pointer to int

    *ptr = 15;    // ? Allowed: can modify the value
    // ptr = &b;  // ? Not allowed: cannot point to another address

    cout << "Updated value pointed by ptr: " << *ptr << endl;
    return 0;
}
```

The const Keyword

Constant Data Members

- Data members are not initialized during declaration.
- The initialization is done in the constructor
- Once initialized cannot change the value.

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

class Car {
public:
    const int wheels;

    // Constructor with initializer list
    Car(int w) : wheels(w) {}

    void display() {
        cout << "This car has " << wheels << " wheels." << endl;
    }
};

int main() {
    Car car1(4);
    car1.display();

    // car1.wheels = 5; // [Error] assignment of read-only member 'Car::wheels'

    return 0;
}
```

The const Keyword

Constant Objects

- Data members can never be changed

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

class Car {
private:
    int speed;

public:
    Car(int s) : speed(s) {}

    int getSpeed() const {
        return speed;
    }

    // This function is not const, so it cannot be called by a const object
    void setSpeed(int s) {
        speed = s;
    }
};

int main() {
    const Car myCar(100);  // constant object

    cout << "Speed: " << myCar.getSpeed() << endl;

    // myCar.setSpeed(120);  // ? Error: can't call non-const member function

    return 0;
}
```

The this Keyword

What is this ?

- A pointer that points to the current object of a class.
- Automatically available in non-static member functions.
- Useful when:
  - Parameter names conflict with member variables.
  - You want to return the current object.
  - Working with method chaining.

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

class Car {
    string brand;
    int year;

public:
    void setDetails(string brand, int year) {
        this->brand = brand;  // Resolves naming conflict
        this->year = year;
    }

    void display() {
        cout << "Brand: " << this->brand << ", Year: " << this->year << endl;
    }
};

int main() {
    Car car1;
    car1.setDetails("Toyota", 2020);
    car1.display();  // Output: Brand: Toyota, Year: 2020
    return 0;
}
```

Function Overloading

What is Function Overloading?

- Function overloading allows multiple functions with the same name but different parameter lists.
- It enables compile-time polymorphism.
- Improves code readability and reusability.

- Overloaded functions must differ in their **parameter lists**
- Must have either
  - different numbers of parameters
  - different types of parameters.
- Functions can not be overloaded if they differ only in the **return type.**

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

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

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

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


int main() {
    cout << add(3, 4) << endl;          // Output: 7
    cout << add(2.5, 4.3) << endl;      // Output: 6.8
    cout << add(1, 2, 3) << endl;       // Output: 6
    return 0;
}
```

Operator Overloading

What is Operator Overloading?

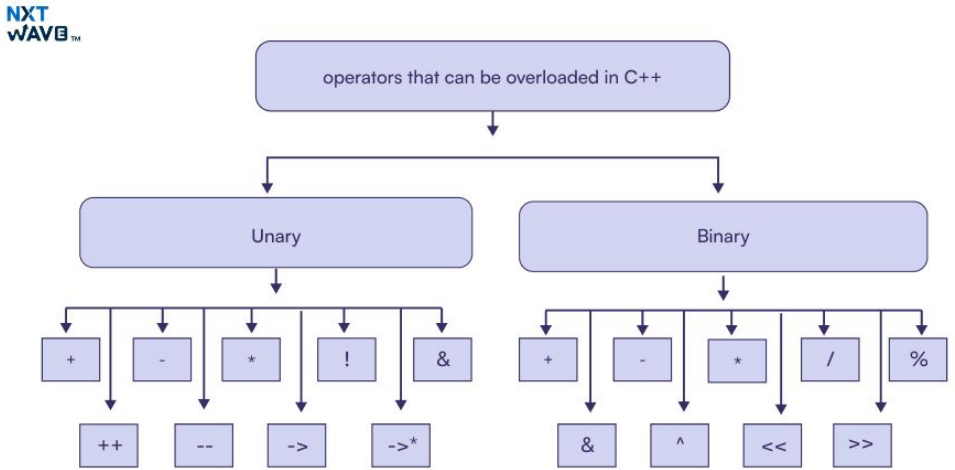
- Allows custom behavior for operators when used with user-defined types (e.g., classes).
- Improves code readability and intuitive use of objects.

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

class Point {
    int x, y;

public:
    Point(int xVal, int yVal) : x(xVal), y(yVal) {}

    // Overload the + operator
    Point operator + (const Point& other) {
        return Point(x + other.x, y + other.y);
    }

    void display() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Point p1(2, 3);
    Point p2(4, 5);

    Point p3 = p1 + p2;  // operator+ is called
    p3.display();        // Output: (6, 8)

    return 0;
}
```

Operator Overloading

Operators that can NOT be Overloaded

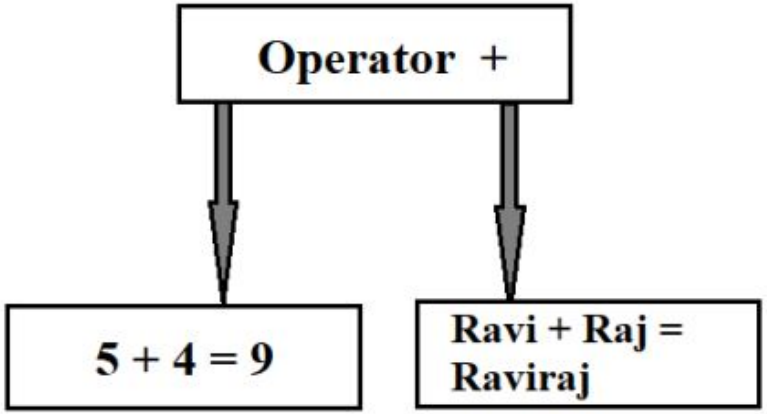
- The dot operator (.)
- The scope resolution operator (::)
- The dereferencing operator (*)
- The ternary conditional operator (? :)
- The sizeof operator



Operator Overloading

Limitations of operator overloading

- Some operators are not overloadable
- Cannot Change Precedence or Associativity
- Cannot Create New Operators
- Overuse Can Reduce Readability
- At Least One Operand Must Be a User-Defined Type
    - You cannot overload an operator for two primitive types like int + float.



Operator Overloading

Approaches for operator overloading

- Overloading unary operator.
- Overloading binary operator.
- Overloading operators using a friend function

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

class Point {
    int x, y;

public:
    Point(int xVal, int yVal) : x(xVal), y(yVal) {}

    // Overload the + operator
    Point operator + (const Point& other) {
        return Point(x + other.x, y + other.y);
    }

    void display() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Point p1(2, 3);
    Point p2(4, 5);

    Point p3 = p1 + p2;  // operator+ is called
    p3.display();        // Output: (6, 8)

    return 0;
}
```

Operator Overloading

Overloading Unary Operator

- Operates on one operand (e.g., -obj, ++obj).
- Can be overloaded as member or friend.

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

class Count {
    int value;
public:
    Count(int v) : value(v) {}

    void operator++() {  // Unary operator++
        ++value;
    }

    void display() {
        cout << "Value: " << value << endl;
    }
};

int main() {
    Count c(5);
    c.display(); // Output: Value: 5
    ++c;  // Calls overloaded operator++
    c.display();  // Output: Value: 6
    return 0;
}
```

Operator Overloading

Overloading Binary Operator

- Operates on two operands (e.g., obj1 + obj2).
- Often overloaded as member or friend.

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

class Point {
    int x;
public:
    Point(int val) : x(val) {}

    Point operator+(const Point& p) {  // Binary operator+
        return Point(x + p.x);
    }

    void display() {
        cout << "X: " << x << endl;
    }
};

int main() {
    Point p1(3), p2(7);
    Point p3 = p1 + p2;  // Calls operator+
    p3.display();        // Output: X: 10
    return 0;
}
```

Operator Overloading

Overloading Using Friend Function

- Allows operator function to access private members.
- Useful when left operand isn't a class object.

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

class Distance {
    int meters;
public:
    Distance(int m) : meters(m) {}

    // Declare friend function
    friend Distance operator+(const Distance&, const Distance&);

    void display() {
        cout << "Meters: " << meters << endl;
    }
};

// Define friend function
Distance operator+(const Distance& d1, const Distance& d2) {
    return Distance(d1.meters + d2.meters);
}

int main() {
    Distance d1(20), d2(30);
    Distance d3 = d1 + d2;  // Calls friend operator+
    d3.display();           // Output: Meters: 50
    return 0;
}
```