Notes
Operator Overloading

What is Operator Overloading?

Operator overloading allows us to **redefine the behavior of operators** (+, -, *, =, etc.) for **user-defined types** (classes).

This makes operations on objects more **intuitive and readable**.

Why Overload Operators?

- Allows **natural operations** on objects (e.g., p1 + p2 instead of p1.add(p2)).
- Makes **object-oriented programming (OOP) cleaner**.
- Improves **code readability and maintainability**.

Operator Overloading

Rules for Operator Overloading

- Only **existing operators** can be overloaded.
- **Precedence and associativity** of operators do not change.
- Some operators **must be member functions** (=, (), [], **->**).
- Other operators can be **member functions or global functions**.
- Cannot overload ::, .*, sizeof, typeid, etc.

Operator Overloading

Syntax for Operator Overloading

Operators can be overloaded:

1. **As a member function** (operates on this object).

2. **As a global function** (operates on objects passed as arguments).

Operator Overloading

General Syntax (Member Function)

```
class ClassName {
public:
    ReturnType operator OpSymbol (const ClassName& obj) {
        // Define operation
    }
};
```

General Syntax (Global Function)

```
ReturnType operator OpSymbol (const ClassName& obj1, const ClassName& obj2) {
    // Define operation
}
```

Overloading Operators in a Simple Point Class

```
class Point {
private:
    int x, y;

public:
    // Constructor
    Point(int a = 0, int b = 0) : x(a), y(b) {}

    // Overloading + operator (member function)

    Point operator + (const Point& p) const {
        return Point(x + p.x, y + p.y);
    }

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

    // Getter functions for accessing private members
    int getX() const { return x; }
    int getY() const { return y; }
};
```

```
// Overloading - operator (global function)
// Global function for – operator

Point operator - (const Point& p1, const Point& p2) {
    return Point(p1.getX() - p2.getX(), p1.getY() - p2.getY());
}

int main() {
    Point p1(3, 4), p2(1, 2);

    Point p3 = p1 + p2;  // Uses member function
    Point p4 = p1 - p2;  // Uses global function

    cout << "p1: "; p1.display();
    cout << "p2: "; p2.display();
    cout << "p1 + p2: "; p3.display();
    cout << "p1 - p2: "; p4.display();

    return 0;
}
```

```
Point operator+(const Point& p) const {
    return Point(x + p.x, y + p.y);
}
```

```
Point operator+(const Point& p) const {
    Point temp;
    temp.x = x + p.x;
    temp.y = y + p.y;
    return temp;
}
```

```
Point operator+(const Point& p) const {
    return Point(this->x + p.x, this->y + p.y);
}
```

```
Point operator+(const Point& p) const {
    return Point((x > 0 ? x + p.x : x), (y > 0 ? y + p.y : y));
}
```

Overloading ++ and -- (Pre/Post Increment)

```
// Pre-increment (++p)

Point& operator ++ () {
    x++; y++;
    return *this;
}
```

```
// Post-increment (p++)

Point operator ++ (int) {
    Point temp = *this;
    x++; y++;
    return temp;
}
```

Overloading == (Equality Comparison)

```
bool operator == (const Point& p) const {
    return (x == p.x && y == p.y);
}
```

Overloading << and >> (I/O Stream Operators)

```
ostream& operator<<(ostream& out, const Point& p) {
    out << "(" << p.getX() << ", " << p.getY() << ")";
    return out;
}

istream& operator>>(istream& in, Point& p) {
    int a, b;
    in >> a >> b;
    p = Point(a, b);
    return in;
}
```

Practice Questions

1. Overload the * operator for scalar multiplication (p * 2).
2. Overload the / operator for division by a scalar.
3. Implement a Complex class with overloaded +, -, and * operators.
4. Overload the [] operator for accessing Point coordinates (p[0] for x, p[1] for y).
5. Overload == and != to compare two Point objects.

Operator	**Purpose**
+	Adds two objects (p1 + p2)
-	Subtracts two objects (p1 - p2)
*	Multiplies object with scalar (p1 * 2)
/	Divides object by scalar (p1 / 2)
[]	Accesses elements (p[0], p[1])
==	Checks equality (p1 == p2)
!=	Checks inequality (p1 != p2)

1. Overloading * Operator for Scalar Multiplication (p * 2)

```
class Point {
private:
    int x, y;

public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}

    // Overloading * operator for scalar multiplication
    Point operator*(int scalar) const {
        return Point(x * scalar, y * scalar);
    }

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

```
int main() {
    Point p1(3, 4);
    Point p2 = p1 * 2; // Multiply point by 2

    cout << "p1: "; p1.display();
    cout << "p1 * 2: "; p2.display();

    return 0;
}
```

```
p1: (3, 4)
p1 * 2: (6, 8)
```

2. Overloading / Operator for Division by a Scalar (p / 2)

```
class Point {
private:
    int x, y;

public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}

    // Overloading / operator for scalar division
    Point operator/(int scalar) const {
        if (scalar == 0) {
            cout << "Error: Division by zero!" << endl;
            return *this;
        }
        return Point(x / scalar, y / scalar);
    }

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

```
int main() {
    Point p1(10, 20);
    Point p2 = p1 / 2; // Divide point by 2

    cout << "p1: "; p1.display();
    cout << "p1 / 2: "; p2.display();

    return 0;
}
```

```
p1: (10, 20)
p1 / 2: (5, 10)
```

3. Implementing a Complex Class with Overloaded +, -, and * Operators

```
class Complex {
private:
   double real, imag;

public:
   Complex(double r = 0, double i = 0) : real(r), imag(i) {}

   // Overloading + operator
   Complex operator+(const Complex& c) const {
      return Complex(real + c.real, imag + c.imag);
   }

   // Overloading - operator
   Complex operator-(const Complex& c) const {
      return Complex(real - c.real, imag - c.imag);
   }

   // Overloading * operator
   Complex operator*(const Complex& c) const {
      return Complex(real * c.real - imag * c.imag, real * c.imag + imag * c.real);
   }

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

```
int main() {
   Complex c1(3, 4), c2(1, 2);

   Complex sum = c1 + c2;
   Complex diff = c1 - c2;
   Complex prod = c1 * c2;

   cout << "c1: "; c1.display();
   cout << "c2: "; c2.display();
   cout << "c1 + c2: "; sum.display();
   cout << "c1 - c2: "; diff.display();
   cout << "c1 * c2: "; prod.display();

   return 0;
}
```

```
c1: 3 + 4i
c2: 1 + 2i
c1 + c2: 4 + 6i
c1 - c2: 2 + 2i
c1 * c2: -5 + 10i
```

4. Overloading [] Operator for Accessing Point Coordinates (p[0] for x, p[1] for y)

```
class Point {
private:
    int x, y;

public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}

    // Overloading [] operator
    int operator[](int index) const {
        if (index == 0) return x;
        if (index == 1) return y;
        cout << "Error: Invalid index!" << endl;
        return -1;
    }

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

```
int main() {
    Point p1(5, 10);
    cout << "p1[0]: " << p1[0] << endl;
    cout << "p1[1]: " << p1[1] << endl;
    cout << "p1[2]: " << p1[2] << endl; // Invalid index

    return 0;
}
```

```
p1[0]: 5
p1[1]: 10
Error: Invalid index!
p1[2]: -1
```

5. Overloading == and != to Compare Two Point Objects

```
class Point {
private:
    int x, y;

public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}

    // Overloading == operator
    bool operator==(const Point& p) const {
        return (x == p.x && y == p.y);
    }

    // Overloading != operator
    bool operator!=(const Point& p) const {
        return !(*this == p); // Uses the overloaded ==
    }

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

```
int main() {
    Point p1(3, 4), p2(3, 4), p3(5, 6);

    cout << "p1 == p2: " << (p1 == p2) << endl; // True (1)
    cout << "p1 == p3: " << (p1 == p3) << endl; // False (0)
    cout << "p1 != p3: " << (p1 != p3) << endl; // True (1)

    return 0;
}
```

```
p1 == p2: 1
p1 == p3: 0
p1 != p3: 1
```

1. Operators That Can Be Overloaded

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment (pre/post)
--	Decrement (pre/post)
==	Equality check
!=	Inequality check
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Assignment

Operator	Description
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
<<	Output stream (cout <<)
>>	Input stream (cin >>)
&	Address-of (not recommended to overload)
^	Bitwise XOR
&=	Bitwise AND assignment
^=	Bitwise XOR assignment
~	Bitwise NOT
&&	Logical AND
!	Logical NOT
,	Comma operator
->	Member access (rarely overloaded)
->*	Pointer to member
()	Function call operator
[]	Subscript operator

2. Operators That Cannot Be Overloaded

Operator	Description
.	Member access (object.member)
.*	Pointer-to-member selection
::	Scope resolution (ClassName::member)
sizeof	Size operator (sizeof(type))
typeid	Runtime type identification
alignof	Alignment requirement
new and delete	Memory allocation/deallocation (can be overloaded globally but not per class)

```
class Point {
private:
    double x;
    double y;
public:
    Point(double xVal = 0.0 , double yVal = 0.0) : x(xVal), y(yVal) {}
    // Getter for x
    double getX() const {
        return x;
    }
    // Setter for x
    void setX(double newX) {
        x = newX;
    }
    // Getter for y
    double getY() const {
        return y;
    }
    // Setter for y
    void setY(double newY) {
        y = newY;
    }
   // Display
    void display() {
        cout << "Point(" << x << ", " << y << ")" <<endl;
    }
};
```

```
// Overload + operator for Point + Point
Point operator+(const Point& p1, const Point& p2) {
    Point p;
    p.setX(p1.getX() + p2.getX());
    p.setY(p1.getY() + p2.getY());
    return p;
}

// Overload + operator for Point + scalar
Point operator+(const Point& p, const double scalar) {
    Point r;
    r.setX(p.getX() + scalar);
    r.setY(p.getY() + scalar);
    return r;
}

// Overload + operator for scalar + Point
Point operator+(const double scalar, const Point& p) {
    Point r;
    r.setX(scalar + p.getX());
    r.setY(scalar + p.getY());
    return r;
}
```

```
int main() {
    Point P1(2.0, 3.0);
    Point P2(1.0, 4.0);

    // P1 + P2
    Point result1 = P1 + P2;
    cout << "P1 + P2 = ";
    result1.display();

    // P1 + 5.0
    Point result2 = P1 + 5.0;
    cout << "P1 + 5.0 = ";
    result2.display();

    // 5.0 + P1
    Point result3 = 5.0 + P1;
    cout << "5.0 + P1 = ";
    result3.display();

    return 0;
}
```

```
// Overload + operator for Point + Point
Point operator+(const Point& p2) {
    Point p;
    p.x = (this->x + p2.x);
    p.y = (this->y + p2.y);
    return p;
}

// Overload + operator for Point + scalar
Point operator+(const double scalar) {
    Point r;
    r.x = (this->x + scalar);
    r.y = (this->x + scalar);
    return r;
}
};

// Overload + operator for scalar + Point
Point operator+(const double scalar, const Point& p) {
    Point r;
    r.setX(scalar + p.getX());
    r.setY(scalar + p.getY());
    return r;
}
```

```
int main() {
    Point P1(2.0, 3.0);
    Point P2(1.0, 4.0);

    // P1 + P2
    Point result1 = P1 + P2;
    cout << "P1 + P2 = ";
    result1.display();

    // P1 + 5.0
    Point result2 = P1 + 5.0;
    cout << "P1 + 5.0 = ";
    result2.display();

    // 5.0 + P1
    Point result3 = 5.0 + P1;
    cout << "5.0 + P1 = ";
    result3.display();

    return 0;
}
```

```
// Overload pre-increment operator (++Point)
    Point& operator++() {
        x++;
        y++;
        return *this;
    }
    // Overload post-increment operator (Point++)
    Point operator++(int) {
        Point temp(*this);
        x++;
        y++;
        return temp;
    }
    // Overload pre-decrement operator (--Point)
    Point& operator--() {
        x--;
        y--;
        return *this;
    }
    // Overload post-decrement operator (Point--)
    Point operator--(int) {
        Point temp(*this);
        x--;
        y--;
        return temp;
    }
};
```

```
int main() {
    Point P1(2.0, 3.0);
    cout << "Original Point: ";
    P1.display();

// Pre-increment operator (++Point)
    ++P1;
    cout << "After Pre-increment: ";
    P1.display();

// Post-increment operator (Point++)
    Point result1 = P1++;
    cout << "After Post-increment: ";
    P1.display();
    cout << "Result of Post-increment: ";
    result1.display();

// Pre-decrement operator (--Point)
    --P1;
    cout << "After Pre-decrement: ";
    P1.display();

 // Post-decrement operator (Point--)
    Point result2 = P1--;
    cout << "After Post-decrement: ";
    P1.display();
    cout << "Result of Post-decrement: ";
    result2.display();

    return 0;
}
```

```
// Overload unary + operator to return the point itself
    Point operator+() const {
        return *this;
    }

// Overload unary - operator to negate the point
    Point operator-() const {
        x = -x;
        y = -y;
        return *this;
    }

// Overload unary ! operator to reverse the point's sign
    Point operator!() const {
        x = !x;
        y = !y;
        return *this;
    }
};
```

```
int main() {
    Point P1(2.0, 3.0);

    // Display the original point
    cout << "Original Point: ";
    P1.display();

    // Use unary + operator
    Point result1 = +P1;
    cout << "Unary + Result: ";
    result1.display();

    // Use unary - operator
    Point result2 = -P1;
    cout << "Unary - Result: ";
    result2.display();

    // Use unary ! operator
    Point result3 = !P1;
    cout << "Unary ! Result: ";
    result3.display();

    return 0;
}
```

```
// Overload == operator to compare two Point objects for equality

   bool operator==(const Point& other) const {
      bool areEqual = (x == other.x) && (y == other.y);
      return areEqual;
}

   // Overload != operator to compare two Point objects for inequality

   bool operator!=(const Point& other) const {
      bool areNotEqual = !(*this == other);
      return areNotEqual;
}
};
```

```
int main() {
  Point P1(2.0, 3.0);
  Point P2(2.0, 3.0);
  Point P3(4.0, 5.0);

  if (P1 == P2)
     {
     cout<<"P1 and P2 are equal"<<endl;
      }
     else
     {
     cout<<"P1 and P2 are not equal"<<endl;
      }

  if (P1 != P3)
     {
     cout<<"P1 and P3 are not equal"<<endl;
     }
      else
     {
     cout<<"P1 and P3 are equal"<<endl;
      }
     return 0;
}
```

```
// Overload << operator for easy printing of Point objects
    friend ostream& operator<<(ostream& os, const Point& point) {
        os << "Point(" << point.x << ", " << point.y << ")";
        return os;
    }

    // Overload >> operator to read Point objects from the input stream
    friend istream& operator>>(istream& is, Point& point) {
        is >> point.x >> point.y;
        return is;
    }

// Overload the copy assignment operator (=) for Point objects
    Point& operator=(const Point& other) {
        if (this != &other) {
            x = other.x;
            y = other.y;
        }
        return *this;
    }
};
```

```
int main() {
    Point P1(2.0, 3.0);
    Point P2(0.0, 0.0);

    // Output P1
    cout << "P1: " << P1 <<endl;

    // Input P2 from the user
    cout << "Enter coordinates for P2 (x y): ";
    cin >> P2;

    // Output P2
    cout << "P2: " << P2 <<endl;

// Assign P1 to P2 using the copy assignment operator
    P2 = P1;

    // Output P2 (now contains the same values as P1)
    cout << "P2: " << P2 << endl;
    return 0;
}
```

In the context of the Point class, we've covered many of the commonly overloaded operators, including:

Arithmetic operators (+, -, *, /) for addition, subtraction, multiplication, and division.
Comparison operators (== and !=) for equality and inequality.
Output stream operator (<<) for printing Point objects.
Input stream operator (>>) for reading Point objects from the input.
Copy assignment operator (=) for copying one Point object to another.
Unary operators (+, -, !, ++, --) for various operations.

Extra, try yourself as **home tasks:**
+= and -= operators: You can overload these operators to perform addition and subtraction assignment.

*, /, %, etc., for more advanced arithmetic operations: Depending on your use case, you might need to provide custom behaviors for these operators.

Relational operators (<, >, <=, >=): If you need custom behavior when comparing Point objects, you can overload these operators.

[] operator: If your class represents a container or has array-like behavior, you can overload the subscript operator for element access.

() operator: If your class is callable like a function, you can overload the function call operator.