



# OBJECT PROGRAMMING

- LECTURE 7 -

(1<sup>ST</sup> YEAR OF STUDY)

# Contents

2

## 7. Operator Overloading

7.1. Operator Overloading in C++

7.2. Types of Operator Overloading

7.3. Operators that can be Overloaded

# 7.1. Operator Overloading in C++

3

- In C++, **Operator Overloading** is a compile-time polymorphism. It is an ability of giving special meaning to an existing operator in C++, without changing its original meaning.
- We will further discuss about operator overloading in C++ with examples, and see which operators we can or cannot overload in C++.
- For example, we can overload an operator ‘+’ in a class like String, so that we can concatenate two strings by just using +.

# 7.1. Operator Overloading in C++

4

- Other example classes where arithmetic operators may be overloaded are Complex numbers, Fractional numbers, Big integers, etc.

```
int a;  
float b, sum;  
sum = a + b;
```

- Here, variables “a” and “b” are of types “int” and “float”, which are built-in data types. Hence the addition operator ‘+’ can easily add the contents of “a” and “b”. This is because the *addition operator* “+” is predefined to add variables of built-in data type only.

# 7.1. Operator Overloading in C++

5

- In this example, we have 3 variables “a1”, “a2” and “a3” of type “class A”. Here we are trying to add two objects “a1” and “a2”, which are of user-defined type, i.e., of type “class A” using the “+” operator.
- This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types.

```
// C++ Program to Demonstrate the
// working/Logic behind Operator
// Overloading
class A {
    statements;
};

int main()
{
    A a1, a2, a3;
    a3 = a1 + a2;
    return 0;
}
```

# 7.1. Operator Overloading in C++

6

- But here, “class A” is a user-defined type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.
- Now, if the user wants to make the operator “+” add two class objects, the user has to *redefine* the meaning of the “+” operator, such that it adds two class objects. This is done by using the concept of “Operator overloading”.
- Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

# 7.1. Operator Overloading in C++

7

```
// C++ Program to Demonstrate
// Operator Overloading
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

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

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void print() { cout << real << " + " << imag << '\n'; }
};
```

```
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output

12 + i9

# 7.1. Operator Overloading in C++

8

- **Operator functions** are the same as normal functions.
- The only differences are, that the name of an operator function is always the **operator keyword** followed by the **symbol** of the operator, and operator functions are called when the corresponding operator is used.

# 7.1. Operator Overloading in C++

9

```
#include <iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }
    void print() { cout << real << " + " << imag << endl; }
    // The global operator function is made friend of this
    // class so that it can access private members
    friend Complex operator+(Complex const& c1,
                            Complex const& c2);
};

Complex operator+(Complex const& c1, Complex const& c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3
        = c1
        + c2; // An example call to "operator+"
    c3.print();
    return 0;
}
```

Output

12 + i9

# 7.1. Operator Overloading in C++

10

- Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

`sizeof`

`typeid`

`Scope resolution (::)`

`Class member access operators (.(dot), .* (pointer to member operator))`

`Ternary or conditional (?:)`

- We can overload:

- *Unary operators*
- *Binary operators*
- *Special operators ([], (), etc)*

# 7.1. Operator Overloading in C++

11

Operators that can be overloaded	Examples		
Binary Arithmetic	+, -, *, /, %	Subscript	[]
Unary Arithmetic	+, -, ++, —	Function call	()
Assignment	=, +=, *=, /=, -=, %=	Logical	&,   , !
Bitwise	&,  , <<, >>, ~, ^	Relational	>, <, ==, <=, >=
De-referencing	(->)		
Dynamic memory allocation, De-allocation	New, delete		

# 7.1. Operator Overloading in C++

12

- Why can't the **above-stated operators** be overloaded?
- **1. sizeof Operator**
  - This returns the size of the object or datatype entered as the operand.
  - This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using overloading would cause a fundamental part of the language to collapse.

# 7.1. Operator Overloading in C++

13

## • 2. typeid Operator

- This provides a CPP program with the ability to recover the actually derived type of the object referred to by a pointer or reference.
- For this operator, the whole point is to uniquely identify a type. If we want to make a user-defined type ‘look’ like another type, polymorphism can be used but the meaning of the typeid operator must remain unaltered, or else serious issues could arise.

# 7.1. Operator Overloading in C++

14

- **3. Scope resolution (::) Operator**

- This helps identify and specify the context to which an identifier refers by specifying a namespace.
- It is completely evaluated at runtime and works on names rather than values.
- The operands of scope resolution are note expressions with data types and CPP has no syntax for capturing them if it were overloaded. So it is syntactically impossible to overload this operator.

# 7.1. Operator Overloading in C++

15

- **4. Class member access operators (.(dot), .\***  
**(pointer to member operator))**

□ The importance and implicit use of class member access operators can be understood through the following example:

```
// C++ program to demonstrate operator overloading
// using dot operator
#include <iostream>
using namespace std;

class ComplexNumber {
private:
    int real;
    int imaginary;

public:
    ComplexNumber(int real, int imaginary)
    {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() { cout << real << " + i" << imaginary; }
    ComplexNumber operator+(ComplexNumber c2)
    {
        ComplexNumber c3(0, 0);
        c3.real = this->real + c2.real;
        c3.imaginary = this->imaginary + c2.imaginary;
        return c3;
    }
}; int main()
{
    ComplexNumber c1(3, 5);
    ComplexNumber c2(2, 4);
    ComplexNumber c3 = c1 + c2;
    c3.print();
    return 0;
}
```

Output

5 + i9

# 7.1. Operator Overloading in C++

16

## □Explanation:

- The statement `ComplexNumber c3 = c1 + c2;` is internally translated as `ComplexNumber c3 = c1.operator+ (c2);` in order to invoke the operator function.
- The argument `c1` is implicitly passed using the '`.`' operator. The next statement also makes use of the dot operator to access the member function `print` and pass `c3` as an argument.

# 7.1. Operator Overloading in C++

17

## • 5. Ternary or Conditional (?:) Operator

- The **ternary** or **conditional** operator is a shorthand representation of an ***if-else*** statement.
- In the operator, the true/false expressions are only evaluated on the basis of the truth value of the conditional expression.

```
conditional statement ? expression1 (if statement is TRUE) : expression2  
(else)
```

```
ABC operator ?: (bool condition, ABC trueExpr, ABC falseExpr);
```

- A function overloading the ternary operator for a class say ABC using the above definition would not be able to guarantee that only one of the expressions was evaluated. Thus, the ternary operator cannot be overloaded.

# 7.1. Operator Overloading in C++

18

- **Important Points about Operator Overloading**
  - 1) For operator overloading to work, at least one of the operands must be a **user-defined class object**.
  - 2) **Assignment Operator:** Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behavior is the same as the copy constructor).
  - 3) **Conversion Operator:** We can also write conversion operators that can be used to convert one type to another type. Example:

# 7.1. Operator Overloading in C++

19

```
// C++ Program to Demonstrate the working
// of conversion operator
#include <iostream>
using namespace std;
class Fraction {
private:
    int num, den;
public:
    Fraction(int n, int d)
    {
        num = n;
        den = d;
    }
    // Conversion operator: return float value of fraction
    operator float() const
    {
        return float(num) / float(den);
    }
};
```

```
int main()
{
    Fraction f(2, 5);
    float val = f;
    cout << val << '\n';
    return 0;
}
```

Output

0.4

# 7.1. Operator Overloading in C++

20

- Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.
- 4) Any constructor that can be called with a single argument works as a **conversion constructor**, which means it can also be used for implicit conversion to the class being constructed. Example:

# 7.1. Operator Overloading in C++

21

```
// C++ program to demonstrate can also be used for implicit
// conversion to the class being constructed
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    Point(int i = 0, int j = 0)
    {
        x = i;
        y = j;
    }
    void print()
    {
        cout << "x = " << x << ", y = " << y << '\n';
    }
};

int main()
{
    Point t(20, 20);
    t.print();
    t = 30; // Member x of t becomes 30
    t.print();
    return 0;
}
```

Output

```
x = 20, y = 20
x = 30, y = 0
```

## 7.2. Types of Operator Overloading

22

- C++ provides a special function to change the current functionality of some operators within its class, which is often called as operator overloading.
- Operator Overloading is the method by which we can change the function of some specific operators to do some different tasks.

```
Return_Type classname :: operator op(Argument list)
{
    Function Body
} // This can be done by declaring the function
```

- **Return\_Type** is the value type to be returned to another object.
- **operator op** is the function where the operator is a keyword.
- **op** is the operator to be overloaded.

## 7.2. Types of Operator Overloading

23

- Operator Overloading can be done by using **three approaches**, i.e.,
  - Overloading **unary operator**.
  - Overloading **binary operator**.
  - Overloading **binary operator using a friend function**.

## 7.2. Types of Operator Overloading

24

- **Criteria/Rules to Define the Operator Function:**
  - In the case of a **non-static member function**, the binary operator should have only one argument and the unary should not have an argument.
  - In the case of a **friend function**, the binary operator should have only 2 arguments and the unary should have only 1 argument.
  - All the class member objects should be **public**, if operator overloading is implemented.
  - Operators that cannot be overloaded are `.* :: ?:`
  - Operators that cannot be overloaded when declaring that function as friend function are `= () [] ->`.
  - The operator function must be either a non-static (member function) or a friend function.

## 7.2. Types of Operator Overloading

25

- **1. Overloading Unary Operator**

Let us consider overloading (-) unary operator. In the unary operator function, no arguments should be passed. It works only with one class object. It is the overloading of an operator operating on a single operand. **Example:** Assume that class Distance takes two member objects i.e. feet and inches, and creates a function by which the Distance object should decrement the value of feet and inches by 1 (having a single operand of Distance Type).

# 7.2. Types of Operator Overloading

26

```
// C++ program to show unary
// operator overloading
#include <iostream>
using namespace std;

class Distance {
public:
    int feet, inch;

    // Constructor to initialize
    // the object's value
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }

    // Overloading(-) operator to
    // perform decrement operation
    // of Distance object
    void operator-()
    {
        feet--;
        inch--;
        cout << "\nFeet & Inches(Decrement): " <<
            feet << " " << inch;
    }
};

// Driver Code
int main()
{
    Distance d1(8, 9);

    // Use (-) unary operator by
    // single operand
    -d1;
    return 0;
}
```

Output

Feet & Inches(Decrement): 7'8

## 7.2. Types of Operator Overloading

27

**Explanation:** In the above program, it shows that no argument is passed and no return type value is returned, because the unary operator works on a single operand. (-) operator changes the functionality to its member function.

- **2. Overloading Binary Operator**

In the binary operator overloading function, there should be one argument to be passed. It is the overloading of an operator operating on two operands. Below is the C++ program to show the overloading of the binary operator (+) using a class Distance with two distant objects.

## 7.2. Types of Operator Overloading

28

```
// C++ program to show binary  
// operator overloading  
  
#include <iostream>  
using namespace std;  
  
class Distance {  
public:  
    int feet, inch;  
  
    Distance()  
    {  
        this->feet = 0;  
        this->inch = 0;  
    }  
  
    Distance(int f, int i)  
    {  
        this->feet = f;  
        this->inch = i;  
    }  
  
    // Overloading (+) operator to  
    // perform addition of two distance  
    // object  
    // Call by reference
```

```
Distance operator+(Distance& d2)  
{  
    // Create an object to return  
    Distance d3;  
  
    d3.feet = this->feet + d2.feet;  
    d3.inch = this->inch + d2.inch;  
  
    // Return the resulting object  
    return d3;  
}  
};  
  
// Driver Code  
int main()  
{  
    Distance d1(8, 9);  
    Distance d2(10, 2);  
    Distance d3;  
  
    // Use overloaded operator  
    d3 = d1 + d2;  
  
    cout << "\nTotal Feet & Inches: " <<  
        d3.feet << "'" << d3.inch;  
    return 0;  
}
```

Output

Total Feet & Inches: 18'11

## 7.2. Types of Operator Overloading

29

### ***Explanation:***

- **Line 27, Distance operator+(Distance &d2):** Here return type of function is distance and it uses call by references to pass an argument.
- **Line 49, d3 = d1 + d2:** Here, d1 calls the operator function of its class object and takes d2 as a parameter, by which the operator function returns the object and the result will reflect in the d3 object.

## 7.2. Types of Operator Overloading

30

- **3. Overloading Binary Operator using a Friend function**

In this approach, the operator overloading function must be preceded by the **friend** keyword, and declare the function in the class scope. Keeping in mind, the friend operator function takes two parameters in a binary operator and varies one parameter in a unary operator. All the working and implementation would same as the binary operator function except this function will be implemented outside the class scope. **Example:** Below is the C++ program to show binary operator overloading using a friend function.

# 7.2. Types of Operator Overloading

31

```
// C++ program to show binary  
// operator overloading using  
// a Friend Function  
#include <iostream>  
using namespace std;  
  
class Distance {  
public:  
  
    int feet, inch;  
  
    Distance()  
    {  
        this->feet = 0;  
        this->inch = 0;  
    }  
  
    Distance(int f, int i)  
    {  
        this->feet = f;  
        this->inch = i;  
    }  
  
    // Declaring friend function  
    // using friend keyword  
    friend Distance operator + (Distance&,  
        Distance&);  
};  
  
// Implementing friend function  
// with two parameters  
// Call by reference  
Distance operator+(Distance& d1,  
                    Distance& d2)  
{  
    // Create an object to return  
    Distance d3;  
  
    d3.feet = d1.feet + d2.feet;  
    d3.inch = d1.inch + d2.inch;  
  
    // Return the resulting object  
    return d3;  
}  
  
// Driver Code  
int main()  
{  
    Distance d1(8, 9);  
    Distance d2(10, 2);  
    Distance d3;  
  
    // Use overloaded operator  
    d3 = d1 + d2;  
  
    cout << "\nTotal Feet & Inches: " <<  
        d3.feet << "'" << d3.inch;  
    return 0;  
}
```

Output

Total Feet & Inches: 18'11

## 7.2. Types of Operator Overloading

32

- ***Explanation:*** The operator function is implemented outside of the class scope by declaring that function as the friend function.
- In these ways, an operator can be overloaded to perform certain tasks by changing the functionality of operators.

## 7.3. Operators that can be Overloaded

33

- There are various ways to overload Operators in C++ by implementing any of the following types of functions:
  - 1) Member Function**
  - 2) Non-Member Function**
  - 3) Friend Function**

## 7.3. Operators that can be Overloaded

34

- In C++, the programmer abstracts real-world objects using classes as concrete types. Sometimes, it is required to convert one concrete type to another concrete type or primitive type implicitly. **Conversion operators** play an important role in such situations. It is similar to the operator overloading function in class.
- For example consider the following class, here, we are making a class for complex numbers.
- It has two data members:
  - **real**
  - **imaginary**

# 7.3. Operators that can be Overloaded

35

```
// CPP Program to demonstrate Conversion Operators
#include <cmath>
#include <iostream>
using namespace std;

class Complex {
private:
    double real;
    double imag;

public:
    // Default constructor
    Complex(double r = 0.0, double i = 0.0)
        : real(r)
        , imag(i)
    {
    }

    // magnitude : usual function style
    double mag() { return getMag(); }

    // magnitude : conversion operator
    operator double() { return getMag(); }
}
```

```
private:
    // class helper to get magnitude
    double getMag()
    {
        return sqrt(real * real + imag * imag);
    }

int main()
{
    // a Complex object
    Complex com(3.0, 4.0);

    // print magnitude
    cout << com.mag() << endl;
    // same can be done like this
    cout << com << endl;
}
```

Output

```
5
5
```

# 7.3. Operators that can be Overloaded

36

- List of operators that can be overloaded are:

+	-	*	?	%	?	&		~
!	=	<	>	+=	-=	*=	?=	%=
?=	&=	=	<<	>>	<<=	>>=	==	!=
<=	>=	&&		++	--	,	->*	->
()	[]	new	delete	new[]	delete[]			

# 7.3. Operators that can be Overloaded

37

- Example 1: Overloading ++ Operator

```
// CPP program to illustrate
// operators that can be overloaded
#include <iostream>
using namespace std;

class overload {
private:
    int count;

public:
    overload()
        : count(4)
    {
    }

    void operator++() { count = count + 1; }
    void Display() { cout << "Count: " << count; }
};

int main()
{
    overload i;
    // this calls "function void operator ++()" function
    ++i;
    i.Display();
    return 0;
}
```

Output

Count: 5

## 7.3. Operators that can be Overloaded

38

- The increment (++) and decrement (--) operators are unary operators that are commonly used for incrementing and decrementing a value by one.
- Both can be used as pre-increment/-decrement, or post-increment/-decrement operators.
- **Pre-incrementing** a value increments the value, and returns the *incremented* value. **Post-incrementing** a value increments the value, but returns the *original* value.
- It's analogous for decrementing.

## 7.3. Operators that can be Overloaded

39

- Overloading increment/decrement operators.

```
E.g., int x=15;  
cout<<++x; cout<<x; cout<<x++; cout<<x;  
cout<<--x; cout<<x; cout<<x--; cout<<x;  
would output 1616161716161615;
```

- This is the reason why pre-incrementing/-decrementing is preferred to post-incrementing/-decrementing.
- This should be reflected in the code for the increment/decrement operators for the user defined types.

# 7.3. Operators that can be Overloaded

40

- Example 2: Overloading ++ operator, i.e., pre and post increment operator

```
// CPP program to demonstrate the  
// Difference between pre increment  
// and post increment overload operator  
  
#include <iostream>  
using namespace std;  
  
class overload {  
private:  
    int count;  
  
public:  
    overload(int i)  
        : count(i)  
    {  
    }  
  
    overload operator++(int) { return (count++); }  
    overload operator++()  
    {  
        count = count + 1;  
        return count;  
    }  
    void Display() { cout << "Count: " << count << endl; }  
};
```

// Driver code

```
int main()  
{  
    overload i(5);  
    overload post(5);  
    overload pre(5);
```

```
    // this calls "function overload operator ++()" function  
    pre = ++i;  
    cout << "results of I = ";  
    i.Display();  
    cout << "results of preincrement = ";  
    pre.Display();  
    // this call "function overload operator ++()" function  
    i++; // just to show diff  
    i++; // just to show diff  
    post = i++;  
    cout << "Results of post increment = ";  
    post.Display();  
    cout << "And results of i , here we see difference : "  
        " ";  
    i.Display();  
    return 0;
```

Output

```
results of I = Count: 6  
results of preincrement = Count: 6  
Results of post increment = Count: 8  
And results of i , here we see difference : Count: 9
```

# 7.3. Operators that can be Overloaded

41

- Example 3: Overloading [ ] operator

```
// CPP program to illustrate overloading the
// [ ] operator

#include <iostream>
using namespace std;
class overload {
    int a[3];

public:
    overload(int i, int j, int k)
    {
        a[0] = i;
        a[1] = j;
        a[2] = k;
    }
    int operator[](int i) { return a[i]; }
};

int main()
{
    overload ob(1, 2, 3);
    cout << ob[1]; // displays 2
    return (0);
}
```

Output

2

# 7.3. Operators that can be Overloaded

42

- Example 4: Overloading -> operator

```
// CPP program to illustrate
// operators that can be overloaded
#include <bits/stdc++.h>
using namespace std;

class GFG {
public:
    int num;
    GFG(int j) { num = j; }
    GFG* operator->(void) { return this; }
};
```

Output

```
T.num = 5
Ptr->num = 5
T->num = 5
```

```
// Driver code
int main()
{
    GFG T(5);
    GFG* Ptr = &T;

    // Accessing num normally
    cout << "T.num = " << T.num << endl;

    // Accessing num using normal object pointer
    cout << "Ptr->num = " << Ptr->num << endl;

    // Accessing num using -> operator
    cout << "T->num = " << T->num << endl;

    return 0;
}
```

## 7.3. Operators that can be Overloaded

43

- **List of operators that cannot be overloaded**
  - 1) Scope Resolution Operator ()::
  - 2) Ternary or Conditional Operator (?:)
  - 3) Member Access or Dot operator (.)
  - 4) Pointer-to-member Operator (.\* )
  - 5) Object size Operator (sizeof)
  - 6) Object type Operator(typeid)
- The above operators will generate an error, if overloaded.

## 7.3. Operators that can be Overloaded

44

- Example 5: Overloading the .(dot) operator. Dot (.) operator can't be overloaded, so it will generate an error.

```
// C++ program to illustrate
// Overloading this .(dot) operator
#include <iostream>
using namespace std;    Output: Error
```

```
class cantover {
public:
    void fun();
};

class X {
    cantover* p;
    cantover& operator.() { return *p; }
    void fun();
};

void g(X& x)
{
    x.fun(); // X::fun or cantover::fun or error?
}
```

```
prog.cpp:12:23: error: expected type-specifier before ‘.’ token
                cantover& operator.() { return *p; }
```