UNIT –II

Operator Overloading , Inheritance & Poymorphism

Operator Overloading in C++

in C++, Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
In this article, we will further discuss about operator overloading in C++ with examples and see which operators we can or cannot overload in C++.

C++ Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile-time polymorphism. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

Example:

```
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.

Implementation:

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

int main()
{
   A a1, a2, a3;

   a3 = a1 + a2;

   return 0;
}
```

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. 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". So the main idea behind "Operator

overloading" is to use C++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

Example of Operator Overloading in C++

```
// 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 << " + i" << imag << '\n'; }
};

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

Output

```
12 + i9
```

Difference between Operator Functions and Normal Functions

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.

Example

```
#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 << " + i" << 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 &quot;operator+&quot;
    c3.print();
    return 0;
}
```

Output

```
12 + i9
```

Can We Overload All Operators?

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 (?:)
```

Operators that can be Overloaded in C++

We can overload

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

But, among them, there are some operators that cannot be overloaded. They are

- ***Scope resolution operator*** *(::)(::)*
- ***Member selection operator***
- ***Member selection through*** ***

Pointer to a member variable

- **Conditional operator** (?:)(?:)
- **Sizeof operator sizeof()**

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

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.

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.

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.

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:

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
```

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.
Besides, these operators also work on names and not values and there is no provision (syntactically) to overload them.

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)
```

A function overloading the ternary operator for a class say ABC using the definition

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

would not be able to guarantee that only one of the expressions was evaluated. Thus, the ternary operator cannot be overloaded.

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). See this for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

Example:

```
// 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
```

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:

```
// 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
```

Types of Operator Overloading in C++

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 some specific operators' functions to do different tasks.

Syntax:

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

Here,

- **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.

Operator Overloading can be done by using **two approaches**, i.e.

1. Overloading **Unary Operator.**
2. Overloading **Binary Operator**.

Criteria/Rules to Define the Operator Function

1. 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.
2. In the case of a **friend function**, the binary operator should have only two arguments and the unary should have only one argument.
3. Operators that cannot be overloaded are **.* :: ?:**
4. Operators that cannot be overloaded when declaring that function as friend function are = () [] ->.
5. The operator function must be either a non-static (member function), global free function or a friend function.

Refer to this, for more rules of Operator Overloading.

Operator overloading allows you to redefine the way operators work with user-defined types. To master the various types of operator overloading in C++, explore the **C++ Course**, which provides comprehensive tutorials and examples.

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).

```
// 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
```

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.
Note: *d2 = -d1 will not work, because operator-() does not return any value.*

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.

```
// 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
```

Explanation:

1. **Line 27, Distance operator+(Distance &d2):** Here return type of function is distance and it uses call by references to pass an argument.
2. **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.

Difference between Inheritance and Polymorphism

Inheritance **i**s one in which a new class is created that inherits the properties of the already exist class. It supports the concept of code reusability and reduces the length of the code in object-oriented programming.

Types of Inheritance are:

1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritances
4. Hybrid inheritance
5. Hierarchical inheritance

Example of Inheritance:

C++JavaC#JavaScriptPython3

```
#include <iostream>using namespace std;
class A {    int a, b;
public:    void add(int x, int y)    {        a = x;        b = y;        cout << "addition of a+b is:" <<
(a + b) << endl;    }};
class B : public A {public:    void print(int x, int y)    {        add(x, y);    }};
int main(){    B b1;    b1.print(5, 6);    return 0;}
```

Output

```
addition of a+b is:11
```

Here, class B is the derived class which inherit the property(**add method**) of the base class A.

Polymorphism**:**

Polymorphism is that in which we can perform a task in multiple forms or ways. It is applied to the functions or methods. Polymorphism allows the object to decide which form of the function to implement at compile-time as well as run-time.

Types of Polymorphism are:

1. Compile-time polymorphism (Method overloading)
2. Run-time polymorphism (Method Overriding)

Example of Polymorphism:

C++Java

```
#include "iostream"using namespace std;
class A {    int a, b, c;
public:    void add(int x, int y)    {        a = x;        b = y;        cout << "addition of a+b is:" << (a + b) << endl;    }
   void add(int x, int y, int z)    {        a = x;        b = y;        c = z;        cout << "addition of a+b+c is:" << (a + b + c) << endl;    }
   virtual void print()    {        cout << "Class A's method is running" << endl;    }};
class B : public A {public:    void print()    {        cout << "Class B's method is running" << endl;    }};
int main(){    A a1;
   // method overloading (Compile-time polymorphism)    a1.add(6, 5);
   // method overloading (Compile-time polymorphism)    a1.add(1, 2, 3);
   B b1;
   // Method overriding (Run-time polymorphism)    b1.print();}
```

Output

```
addition of a+b is:11

addition of a+b+c is:6

Class B's method is running
```

Types of Polymorphism are:

Compile-time polymorphism (Method overloading)

Run-time polymorphism (Method Overriding)

Example of Polymorphism:

```
#include "iostream"

using namespace std;
```

```
class A {
    int a, b, c;

public:
    void add(int x, int y)
    {
        a = x;
        b = y;
        cout << "addition of a+b is:" << (a + b) << endl;
    }

    void add(int x, int y, int z)
    {
        a = x;
        b = y;
        c = z;
        cout << "addition of a+b+c is:" << (a + b + c) << endl;
    }

    virtual void print()
    {
        cout << "Class A's method is running" << endl;
    }
};

class B : public A {
public:
    void print()
    {
        cout << "Class B's method is running" << endl;
    }
};
```

```
int main()
{
    A a1;

    // method overloading (Compile-time polymorphism)
    a1.add(6, 5);

    // method overloading (Compile-time polymorphism)
    a1.add(1, 2, 3);

    B b1;

    // Method overriding (Run-time polymorphism)
    b1.print();
}
```

Output

```
addition of a+b is:11
addition of a+b+c is:6
Class B's method is running
```

Difference between Inheritance and Polymorphism:

S.NO	Inheritance	Polymorphism
1.	Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class).	Whereas polymorphism is that which can be defined in multiple forms.
2.	It is basically applied to	Whereas it is basically

S.NO	Inheritance	Polymorphism
	classes.	applied to functions or methods.
3.	Inheritance supports the concept of reusability and reduces code length in object-oriented programming.	Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding).
4.	Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance.	Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding).
5.	It is used in pattern designing.	While it is also used in pattern designing.
6.	**Example :** The class bike can be inherit from the class of two-wheel vehicles, which is turn could be a subclass of vehicles.	**Example :** The class bike can have method name set_color(), which changes the bike's color based on the name of color you have entered.