UNIT 4:

Operator Overloading

Operator Overloading 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 C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. 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.

Operators that can be overloaded

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	>, < , = =, <=, >=

Unary Operators Overloading in C++

The unary operators operate on a single operand and following are the examples of Unary operators −

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

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

class Distance {
  private:
    int feet;           // 0 to infinite
    int inches;         // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }

    // method to display distance
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches <<endl;
    }

    // overloaded minus (-) operator
    Distance operator- () {
      feet = -feet;
```

```
      inches = -inches;
      return Distance(feet, inches);
    }
};

int main() {
  Distance D1(11, 10), D2(-5, 11);

  -D1;              // apply negation
  D1.displayDistance();   // display D1

  -D2;              // apply negation
  D2.displayDistance();   // display D2

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

F: -11 I:-10
F: 5 I:-11

Binary Operator Overloading in C++

This section will discuss the Binary Operator Overloading in the C++ programming language. An operator which contains two operands to perform a mathematical operation is called the Binary Operator Overloading. It is a polymorphic compile technique where a single operator can perform various functionalities by taking two operands from the programmer or user. There are multiple binary operators like +, -, *, /, etc., that can directly manipulate or overload the object of a class.

For example, suppose we have two numbers, 5 and 6; and overload the binary (+) operator. So, the binary (+) operator adds the numbers 5 and 6 and returns 11. Furthermore, we can also perform subtraction, multiplication, and division operation to use the binary operator for various calculations.

Syntax of the Binary Operator Overloading

Following is the Binary Operator Overloading syntax in the C++ Programming language.

1. return_type :: operator binary_operator_symbol (arg)

```
{
// function definition
}
```

Here,

return_type: It defines the return type of the function.

operator: It is a keyword of the function overloading.

binary_operator_symbol: It represents the binary operator symbol that overloads a function to perform the calculation.

arg: It defines the argument passed to the function.

Steps to Overload the Binary Operator to Get the Sum of Two Complex Numbers

Step 1: Start the program.

Step 2: Declare the class.

Step 3: Declare the variables and their member function.

Step 4: Take two numbers using the user-defined inp()function.

Step 6: Similarly, define the binary (-) operator to subtract two numbers.

Step 7: Call the print() function to display the entered numbers.

Step 8: Declare the class objects x1, y1, sum, and sub.

Step 9: Now call the print() function using the x1 and y1 objects.

Step 10: After that, get the object sum and sub result by adding and subtracting the objects using the '+' and '-' operators.

Step 11: Finally, call the print() and print2() function using the x1, y1, sum, and sub.

Step 12: Display the addition and subtraction of the complex numbers.

Step 13: Stop or terminate the program.

Example 1: Program to perform the addition and subtraction of two complex numbers using the binary (+) and (-) operator

Let's create a program to calculate the addition and subtraction of two complex numbers by overloading the '+' and '-' binary operators in the C++ programming language.

```
1.  /* use binary (+) operator to add two complex numbers. */
2.  #include <iostream>
3.  using namespace std;
4.  class Complex_num
5.  {
6.     // declare data member or variables
7.     int x, y;
8.     public:
9.        // create a member function to take input
10.       void inp()
11.       {
12.          cout << " Input two complex number: " << endl;
13.          cin >> x >> y;
14.       }
15.       // use binary '+' operator to overload
16.       Complex_num operator + (Complex_num obj)
17.       {
18.          // create an object
19.          Complex_num A;
20.          // assign values to object
21.          A.x = x + obj.x;
22.          A.y = y + obj.y;
23.          return (A);
24.       }
25.       // overload the binary (-) operator
26.       Complex_num operator - (Complex_num obj)
27.       {
28.          // create an object
29.          Complex_num A;
```

```
30.          // assign values to object
31.          A.x = x - obj.x;
32.          A.y = y - obj.y;
33.          return (A);
34.       }
35.       // display the result of addition
36.       void print()
37.       {
38.          cout << x << " + " << y << "i" << "\n";
39.       }
40.
41.       // display the result of subtraction
42.       void print2()
43.       {
44.          cout << x << " - " << y << "i" << "\n";
45.       }
46. };
47. int main ()
48. {
49. Complex_num x1, y1, sum, sub; // here we created object of class Addition i.e x1 and y1
50.    // accepting the values
51.    x1.inp();
52.    y1.inp();
53.    // add the objects
54.    sum = x1 + y1;
55.    sub = x1 - y1; // subtract the complex number
56.    // display user entered values
57.    cout << "\n Entered values are: \n";
58.    cout << " \t";
59.    x1.print();
60.    cout << " \t";
61.    y1.print();
62.    cout << "\n The addition of two complex (real and imaginary) numbers: ";
63.    sum.print(); // call print function to display the result of addition
```

```
64.    cout << "\n The subtraction of two complex (real and imaginary) numbers: ";
65.    sub.print2(); // call print2 function to display the result of subtraction
66.    return 0;
67. }
```

Output

```
Input two complex numbers:
5
7
Input two complex numbers:
3
5
Entered values are:
                5 + 7i
                3 + 5i
The addition of two complex (real and imaginary) numbers: 8 + 12i
The subtraction of two complex (real and imaginary) numbers: 2 - 2i
```

In the above program, we take two numbers from the user, and then use the binary operator to overload the '+' and '-' operators to add and subtract two complex numbers in a class.

Arithmetic Operators

Arithmetic Operators in C++ are used to perform arithmetic or mathematical operations on the operands. For example, '**+**' is used for addition, '**–**' is used for subtraction, '*****' is used for multiplication, etc. In simple terms, arithmetic operators are used to perform arithmetic operations on variables and data; they follow the same relationship between an operator and an operand.

C++ Arithmetic operators are of 2 types:

1. **Unary Arithmetic Operator**
2. **Binary Arithmetic Operator**

1. Binary Arithmetic Operator

These operators operate or work with two operands. C++ provides **5** *Binary Arithmetic Operators* for performing arithmetic functions:

Operator	Name of the Operators	Operation	Implementation
+	Addition	Used in calculating the Addition of two operands	x+y

Operator	Name of the Operators	Operation	Implementation
–	Subtraction	Used in calculating Subtraction of two operands	x-y
*	Multiplication	Used in calculating Multiplication of two operands	x*y
/	Division	Used in calculating Division of two operands	x/y
%	Modulus	Used in calculating Remainder after calculation of two operands	x%y

2. Unary Operator

These operators operate or work with a single operand.

Operator	Symbol	Operation	Implementation
Decrement Operator	—	Decreases the integer value of the variable by one	–x or x —
Increment Operator	++	Increases the integer value of the variable by one	++x or x++

Examples: Program to Addition of polar coordinates

Aim: To implement a program to add two polar coordinates using operator overloading.

Code:

```
#define PI 3.14
class polar
{
	double theta;
	double r;
	public:
	void getdata();
	void display();
	double convert(double);
	double revert(double);
	polar operator+(polar p2);
};
void polar::display()
{
	double t;
	cout<<r<<" cos="" "<<theta<<"ø="" +="" "<<r<<"=" sin="" "<<theta<<"ø";<="" p="" style="box-sizing: border-box;"></r<<">
}
double polar::convert(double t)
{
	double x;
	x=(PI/180)*t;
	return(x);
}
```

```
double polar::revert(double t)
{
        double x;
        x=(180*t)/PI;
        return (x);
}

void polar::getdata()
{
        double t;
        cout<<"Enter value of 'r':";
        cin>>r;
        cout<<"Enter value of 'é':";
        cin>>theta;
}

polar polar::operator+(polar p2)
{
  polar p3;
  double x,y,t1,t2,t;
  t1=convert(theta);
  t2=convert(p2.theta);
  x=(r*cos(t1))+(p2.r*(cos(t2)));
  y=(r*sin(t1))+(p2.r*(sin(t2)));
```

```
  t=atan(y/x);
  p3.theta=revert(t);
  p3.r=sqrt((x*x)+(y*y));
  return(p3);
}

void main()
{
        polar p1,p2,p3;
        clrscr();
        p1.getdata();
        p2.getdata();
        cout<<endl<<endl<<"a:";< p="" style="box-sizing: border-box;"></endl<<endl<<"a:";<>
        p1.display();
        cout<<"\nB:";
        p2.display();
        p3=p1+p2;
        cout<<"\n\nAfter addition:\n\nA + B = ";
        p3.display();
        getch();
}
```

Output:

```
Enter value of 'r':1
Enter value of 'Ɵ':45
Enter value of 'r':1
```

Enter value of 'Θ':45

A:1 cos 45° + 1 sin 45°

B:1 cos 45° + 1 sin 45°

After addition:

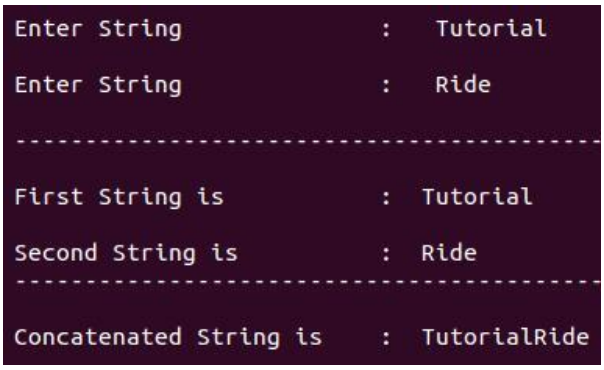
A + B = 2 cos 45° + 2 sin 45°

Concatenate strings using operator overloading

```
#include<iostream>
#include<string.h>
using namespace std;

class String
{
    public:
        char str[20];
    public:
        void accept_string()
        {
            cout<<"\n Enter String          :  ";
            cin>>str;
        }
        void display_string()
        {
            cout<<str;
        }
        String operator+(String x)  //Concatenating String
        {
            String s;
            strcat(str,x.str);
            strcpy(s.str,str);
```

```
            return s;
        }
};
int main()
{
    String str1, str2, str3;

    str1.accept_string();
    str2.accept_string();

    cout<<"\n ----------------------------------------------";
    cout<<"\n\n First String is          :  ";
    str1.display_string();   //Displaying First String

    cout<<"\n\n Second String is         :  ";
    str2.display_string();  //Displaying Second String

    cout<<"\n ----------------------------------------------";
    str3=str1+str2;        //String is concatenated. Overloaded '+' operator
    cout<<"\n\n Concatenated String is    :  ";
    str3.display_string();

    return 0;
}
```

Output:



C++ Comparison Operators

Comparison operators are operators used for comparing two elements, these are mostly used with if-else conditions as they return true-false as result.

There are mainly 6 Comparison Operators namely:

1. **Greater than (>) :** this operator checks whether operand1 is greater than operand2. If the result turns out to be true, it returns true or else returns false. example 5>3 ->returns true
2. **Greater than or equal to (>=) :** this operator checks whether operand1 is greater than or equal to operand2. If the result turns out to be true, it returns true or else returns false. example 5>=5 ->returns true
3. **Less than (<) :** this operator checks whether operand1 is lesser than operand2. If the result turns out to be true, it returns true or else returns false. example 3<5 ->returns true
4. **Less than or equal to (< =) :** this operator checks whether operand1 is lesser than or equal to operand2. If the result turns out to be true, it returns true or else returns false. example 5<=5 ->returns true
5. **Equal to (==) :** this operator checks whether operand1 is equal to operand2. If the result turns out to be true, it returns true or else returns false. example 5==5 ->returns true
6. **Not Equal to (! =) :** this operator checks whether operand1 is not equal to operand2. If the result turns out to be true, it returns true or else returns false. example 5!=3 ->returns true

Comparison Operators have only two return values, either true (1) or False (0).

Data Conversion in C++

A user-defined data types are designed by the user to suit their requirements, the compiler does not support automatic type conversions for such data types therefore, the user needs to design the conversion routines by themselves if required.
There can be 3 types of situations that may come in the data conversion between incompatible data types:

- **Conversion of primitive data type to user-defined type:** To perform this conversion, the idea is to use the constructor to perform type conversion during the object creation.
  - **Conversion of class object to primitive data type:** In this conversion, the **from** type is a class object and the **to** type is primitive data type. The normal form of an overloaded casting operator function, also known as a conversion function. Below is the syntax for the same:

Syntax:

```
operator typename()
{
  // Code
}
```

  - Now, this function converts a **user-defined data type** to a **primitive data type**. For Example, the operator **double()** converts a class object to type double, the operator **int()** converts a class type object to type int, and so on.

Type Conversion in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion** Also known as 'automatic type conversion'.
   - Done by the compiler on its own, without any external trigger from the user.
   - Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
   - All the data types of the variables are upgraded to the data type of the variable with largest data type.
   - bool -> char -> short int -> int ->
   -

- unsigned int -> long -> unsigned ->
-
- long long -> float -> double -> long double
- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:

```
// An example of implicit conversion



#include <iostream>

using namespace std;



int main()

{

   int x = 10; // integer x

   char y = 'a'; // character c



   // y implicitly converted to int. ASCII

   // value of 'a' is 97

   x = x + y;



   // x is implicitly converted to float
```

```
    float z = x + 1.0;



    cout << "x = " << x << endl

         << "y = " << y << endl

         << "z = " << z << endl;



    return 0;

}
```

Output:
x = 107

y = a

z = 108

2. **Explicit Type Conversion**: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type. In C++, it can be done by two ways:

   - **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.
     Syntax:
     (type) expression

     where *type* indicates the data type to which the final result is converted.
     Example:

```
// C++ program to demonstrate

// explicit type casting
```

```
#include <iostream>

using namespace std;



int main()

{

   double x = 1.2;



   // Explicit conversion from double to int

   int sum = (int)x + 1;



   cout << "Sum = " << sum;



   return 0;

}
```

Output:
Sum = 2

- **Conversion using Cast operator:** A Cast operator is an **unary operator** which forces one data type to be converted into another data type.
  C++ supports four types of casting:
  1. Static Cast
  2. Dynamic Cast
  3. Const Cast

4. Reinterpret Cast

Example:

```
#include <iostream>

using namespace std;

int main()

{

   float f = 3.5;



   // using cast operator

   int b = static_cast<int>(f);



   cout << b;

}
```

Output:

3

Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.

Conversions Between Objects and Basic Types

When we want to convert between user-defined data types and basic types, we can't rely on built-in conversion routines, since the compiler d·'esn't know anything about user-defined types besides what we tell it. Instead: we must wu.e these routines ourselves.

Our next example shows how to convert between a basic type and a user-defined type.In this example the user-defined type is (surprise!) the English Distance class from previous examples, and the basic type is float, which we use to represent meters, a unit of length in the metric measurement system.