Unit 3: OVERLOADING, FRIEND FUNCTIONS AND CLASSES

Overloading Definition, Constructor Over-loading, Function Over-loading, drawbacks of functions overloading

Constructor Overloading in C++

Overloaded constructors have the same name (name of the class) but differ in number of arguments. Depending upon the number and type of arguments passed, the corresponding constructor is called.

Example: *Program to demonstrate Constructor Overloading*

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

class Person {
  private:
   int age;
  public:
    Person()  //default constructor
    {
                    age = 45;
                }
    Person(int a) //parameterized constructor, constructor overloaded
    {
                 age = a;
              }
    int getAge() {
                return age;
            }
};
int main() {
   Person person1, person2(55);
   cout << "Person1 Age = " << person1.getAge() << endl;
   cout << "Person2 Age = " << person2.getAge() << endl;
   return 0;
}
```

Output:

Person1 Age =45
Person1 Age =55

Example: *Program to demonstrate constructor overloading*

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

class Room {   private:
    int length;
    intt breadth;
   public:
   Room()    //default constructor
 {
       length = 10;
       breadth = 5;
    }
 //parameterized constructor with 2 parameters, constructor overloaded
   Room(int l, int b)
  {
       length = l;
       breadth = b;
    }
//parameterized constructor with 1 parameter, constructor overloaded
   Room(int len)
  {
       length = len;
       breadth = 10;
    }
    double calculateArea() {
       return length * breadth;
    }
};
int main()
{
    Room room1, room2(12, 6), room3(4);
    cout << "When no argument is passed:" << endl;
    cout << "Area of room = " << room1.calculateArea() << endl;
    cout << "\nWhen (12,6) is passed:" << endl;
    cout << "Area of room = " << room2.calculateArea() << endl;
    cout << "\nWhen breadth is fixed to 10 and (4) is passed:" << endl;
    cout << "Area of room = " << room3.calculateArea() << endl;
    return 0;
}
```

Output:
When no argument is passed:
Area of room = 50
When (12,6) is passed:
Area of room = 72
When breadth is fixed to 10 and (4) is passed:
Area of room = 40

Function Overloading in C++

- In C++, two functions can have the same name if the number and/or type of arguments passed is different.
- These functions having the same name but different arguments are known as overloaded functions.

Example: *Program to demonstrate function overloading*

//area() function is overloaded

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

class FunO
{
  private:
        int s,l,b,r;
  public:
        FunO() {};
        void area(int s)
       {
          this->s=s;
          cout<<"Area of square = "<<s*s<<endl;
       }
       void area(int l,int b)
       {
             this->l=l;
             this->b=b;
             cout<<"Area of rectangle = "<<l*b<<endl;
      }
};
int main() {
         FunO f;
```

```
        f.area(5);
        f.area(4,2);
    return 0;
}
```

Output:
Area of square = 25
Area of rectangle =8

Example: *Program to demonstrate function overloading*
// display() function is overloaded

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

void display(int var1, double var2)
{
   cout << "Integer number: " << var1;
   cout << "and double number: "<< var2 << endl;
}
void display(double var)
{
   cout << "Double number: " << var << endl;
}
void display(int var)
 {
   cout << "Integer number: " << var << endl;
}
int main() {
   int a = 5;
   double b = 5.5;
  display(a);
  display(b);
  display(a, b);
   return 0;
}
```

Output:
Integer number:5
Double number:5.5
Integer number: 5 and double number: 5.5

Benefits of overloaded functions:

1. Readability: Overloaded functions make the code more readable by allowing us to use the same name for similar operations.
2. Flexibility: Overloaded functions provide flexibility by allowing us to perform the same operation on different types of data.
3. Reusability: Overloaded functions promote reusability by reducing the need to write multiple functions with different names for similar operations.
4. Polymorphism: Overloaded functions provide a form of polymorphism by allowing us to use the same name for similar operations on different types of data.

Limitations of overloaded functions:

Overloaded functions can be confusing if not used properly. Here are some limitations to consider when using overloaded functions:

1. Ambiguity: Overloaded functions can be ambiguous if the parameter types are too similar. For example, having overloaded functions with int and long types can cause confusion when passing arguments.
2. Code complexity: Overloading functions can increase code complexity, especially when multiple overloaded functions have to be maintained.
3. Function signatures: Overloading functions can result in function signatures that are difficult to read and understand, especially if the functions have complex parameter lists.

Disadvantages of function overloading:

- Function declarations that differ only in the return type cannot be overloaded

  Example:

  int fun();

  float fun();
- It gives an error because the function cannot be overloaded by return type only.
- Member function declarations with the same name and the same parameter types cannot be overloaded if any of them is a static member function declaration.
- The main disadvantage is that it requires the compiler to perform name mangling on the function name to include information about the argument types.

Operator overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading used to perform the operation on the user-defined types like objects. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

Types of Operator Overloading

- Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading used to perform the operation on the user-defined types like objects.
- For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

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

- Overloading **Unary Operator.**
- Overloading **Binary Operator**.

Operator overloading syntax

```
//Within the class
return_type operator op (arg_list)
{
   //Function body;
}

//Outside the class
return type className :: operator op (arg_list)
{
   //Function body;
}
```

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

Operators that cannot be overloaded

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

Unary Operators Overloading using public member functions

- 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: *Program to demonstrate unary operator (++) overloading*

```
#include <iostream>
using namespace std;
class OperatorOverload {
private:
        int x;
public:
        OperatorOverload() {x=10;}

        void operator ++() {
                x=x + 1;
        }
        void display() {
                cout << "The Count is: " << x;
                }
};
int main() {
        OperatorOverload ov;
        ov++;
        ov.display();
        return 0;
}
```

Output

The Count is 11

Example: *Program to demonstrate unary operator (++) overloading*

```
include <iostream>
using namespace std;
class Counter{
int n;
public:
void read()    {
     cout<<"Enter n value :";
```

```
      cin>>n;
        }
void print()    {
      cout<<n;
        }
void operator ++()    {
          ++n;
     }
};

int main()
{
   Counter obj;
  obj.read();
  ++obj;
 obj.print();
 return 0;
}
```

Output

```
Enter n value 5
6
```

Binary operator overloading using public member functions

- 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.

Example: *Program to demonstrate binary operator (+) overloading*

```
#include <iostream>
using namespace std;
class Demo
{
   int x;
     public:
        Demo(){x=0;}
        Demo(int a)
        {
            x=a;
         }
```

```
    Demo operator + (Demo d)
   {
         Demo result;
         result.x = x+d.x;
           return result;
    }
  void display()
  {
        cout<<"The result of the addition of two objects is: "<<x;
   }
};
int main()
{
   Demo d1(5);
   Demo d2(10);
   Demo d3=d1+d2;
   d3.display();
   return 0;
}
```

Output:
The result of the addition of two objects is: 15

Example: *Program to demonstrate binary operator (+) overloading to add 2 Complex numbers*

```
#include<iostream>
using namespace std;
class ComplexAdd
{
   private:
        int real, imag;
   public:
        ComplexAdd(int r = 0, int i = 0) {
              real = r;
              imag = i;
        }
        ComplexAdd operator + (ComplexAdd ob) {
              ComplexAdd result;
              result.real = real + ob.real;
              result.imag = imag + ob.imag;
              return result;
```

```
        }
        void print() {
                cout << real << " + i" << imag << endl;
        }
};
int main()
{
        ComplexAdd c1(10, 5), c2(4, 2);
        ComplexAdd c3 = c1 + c2;
        c3.print();
}
```

Output:
14 + i 7

Copy Constructor, Assignment Operator Overloading for a Class

Copy Constructor A Copy constructor is an overloaded constructor used to declare and initialize an object from another object.
Default Copy constructor: The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.
User Defined constructor: The programmer defines the user-defined constructor.

Syntax Of User-defined Copy Constructor:

class_name(class_name &old_object);

Example

```
class Ex {
        Ex(Ex &e) // copy constructor.
         { // copyconstructor. }
 } ;
```

Copy constructor can be called in the following ways:

```
  A a2(a1);                    // a1 initializes the a2 objects.
  A a2 = a1;
```

where a1, a2 are objects of class A

Example: *Program to demonstrate copy constructor*

```
#include <iostream>
using namespace std;
class A
{
  public:
    int x;
      A(int a)              // parameterized constructor.
    {
         x=a;
   }
   A(A &i)              // copy constructor
  {
        x = i.x;
  }
};
int main()
{
   A a1(20);             // Calling the parameterized constructor.
   A a2(a1);              //  Calling the copy constructor.
   cout<<a2.x;
   return 0;
}
```

Two types of copies are produced by the constructor:

- Shallow copy
- Deep copy

Shallow Copy

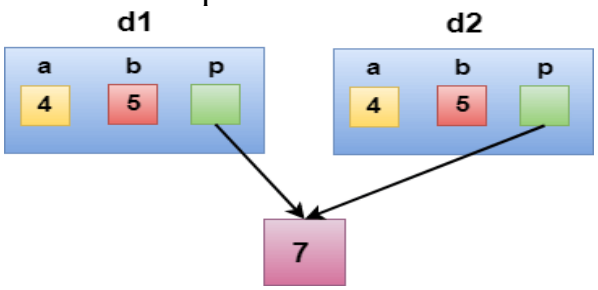
- The default copy constructor can only produce the shallow copy.
- A Shallow copy is defined as the process of creating the copy of an object by copying data of all the member variables as it is.

```
#include <iostream>
  using namespace std;
  class Demo
{
    int a;
    int b;
    int *p;
    public:
    Demo()
    {
```

```
        p=new int;
    }
    void setdata(int x,int y,int z)
    {
        a=x;
        b=y;
        *p=z;
    }
    void showdata()
    {
        cout << "value of a is : " <<a<< endl;
        cout << "value of b is : " <<b<< endl;
        cout << "value of *p is : " <<*p<<endl;
    }
};
int main()
{
  Demo d1;
  d1.setdata(4,5,7);
  Demo d2 = d1;
  d2.showdata();
    return 0;
}
```

Output:

value of a is : 4
value of b is : 5
value of *p is : 7



In the above case, a programmer has not defined any constructor, therefore, the statement **Demo d2 = d1;** calls the default constructor defined by the compiler. The default constructor creates the exact copy or shallow copy of the existing object. Thus, the pointer p of both the objects point to the same memory location. Therefore, when the memory of a field is freed, the memory of another field is also automatically freed as both the fields point to the same memory location. This problem is solved by the **user-defined constructor** that creates the **Deep copy**.

Deep copy

Deep copy dynamically allocates the memory for the copy and then copies the actual value, both the source and copy have distinct memory locations. In this way, both the source and copy are distinct and will not share the same memory location. Deep copy requires us to write the user-defined constructor.

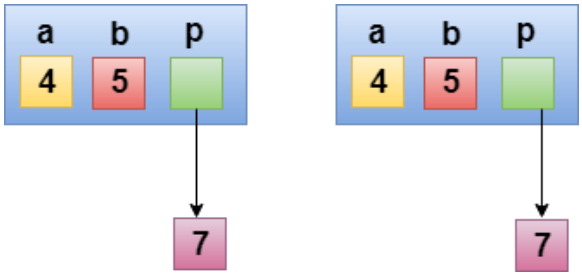
```
#include <iostream>
using namespace std;
class Demo
{
   public:
   int a;
   int b;
   int *p;

   Demo()
   {
      p=new int;
   }
   Demo(Demo &d)
   {
      a = d.a;
      b = d.b;
      p = new int;
      *p = *(d.p);
   }
   void setdata(int x,int y,int z)
   {
      a=x;
      b=y;
      *p=z;
   }
   void showdata()
   {
      cout << "value of a is : " <<a<< endl;
      cout << "value of b is : " <<b<< endl;
      cout << "value of *p is : " <<*p<<endl;
   }
};
int main()
{
  Demo d1;
  d1.setdata(4,5,7);
```

```
  Demo d2 = d1;
  d2.showdata();
  return 0;
}
```

Output:

value of a is : 4
value of b is : 5
value of *p is : 7



In the above case, a programmer has defined its own constructor, therefore the statement **Demo d2 = d1;** calls the copy constructor defined by the user. It creates the exact copy of the value types data and the object pointed by the pointer p. Deep copy does not create the copy of a reference type variable.

Assignment Operator overloading

The assignment operator, "=", is the operator used for Assignment. It copies the right value into the left value. Assignment Operators are predefined to operate only on built-in Data types.

- Assignment operator overloading is binary operator overloading.
- Overloading assignment operator in C++ copies all values of one object to another object.
- Only a non-static member function should be used to overload the assignment operator.

In C++, the compiler automatically provides a default assignment operator for classes. This operator performs a **shallow copy** of each member of the class from one object to another.

This means that if we don't explicitly overload the assignment operator, the compiler will still allow us to assign one object to another using the assignment operator (=), and it won't generate an error.

When we should perform assignment operator overloading?

When our class involves dynamic memory allocation (e.g., pointers) and we need to perform a deep copy to prevent issues like double deletion or data corruption.

Example: *Program to demonstrate assignment operator "=" overloading*

```
#include <iostream>
using namespace std;
class Complex
{
   public:
        int real, img; // real, imaginary
    Complex(int r, int i)
    {
         real = r;
         img = i;
    }
    void operator  =  ( Complex& C)
    {
              real = C.real;
             img = C.img;
    }
    void print()
    {
         cout << real << "+i" << img << endl;
    }
};
int main()
{
     Complex C1(2, 3), C2(4, 6);
     cout << "BEFORE OVERLOADING ASSIGNMENT OPERATOR"<< endl;
     cout << "C1 complex number: ";
     C1.print();
     cout << "C2 complex number: ";
     C2.print();
     C1 = C2; // overloading operator '='
     cout << "AFTER OVERLOADING ASSIGNMENT OPERATOR" << endl;
     cout << "C1 complex number: ";
     C1.print();
     cout << "C2 complex number: ";
     C2.print();
    return 0;
}
```

Output:

```
BEFORE OVERLOADING ASSIGNMENT OPERATOR
```

C1 complex number: 2+i3
C2 complex number: 4+i6
AFTER OVERLOADING ASSIGNMENT OPERATOR
C1 complex number: 4+i6
C2 complex number: 4+i6

Friend Functions, Friend Classes

Friend function

- A friend function of a class is defined outside that class scope but it has the right to access all private and protected members of the class.
- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
- A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

Syntax

friend return_type function_name (arguments); // for a global function

friend return_type class_name::function_name (arguments);
// for a member function of another class

- A friend function is a special function in C++ that in spite of not being a member function of a class has the privilege to **access** the **private and protected data** of a class.
- A friend function is a non-member function or ordinary function of a class, which is declared as a friend using the keyword "**friend**" inside the class. By declaring a function as a friend, all the access permissions are given to the function.
- The keyword "friend" is placed only in the function declaration of the friend function and **not** in the **function definition or call.**
- A friend function is called like an ordinary function. It cannot be called using the object name and dot operator. However, it may accept the object as an argument whose value it wants to access.
- A friend function can be declared in any section of the class i.e. public or private or protected.

Global Function as Friend Function

Example: *Program to demonstrate friend function*

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

class Box {
   double width;

   public:
      friend void printWidth( Box box );
      void setWidth( double w)
      {
          width =w;
      }
};

void printWidth( Box box ) {
   //Because printWidth() is a friend of Box, it can directly access any member of
this class
   cout << "Width of box : " << box.width <<endl;
}

int main() {
   Box box;

   // set box width without member function
   box.setWidth(10.0);

   // Use friend function to print the wdith.
   printWidth( box );

   return 0;
}
```

Output:
Width of box: 10

A function can be friend to any number of classes

Example: *Program to demonstrate friend function which is friend of multiple classes*

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

// Forward declaration
class ABC;

class XYZ {
    int x;
  public:
    void set_data(int a)
    {
        x = a;
    }

    friend void max(XYZ, ABC);
};

class ABC {
    int y;
  public:
    void set_data(int a)
    {
      y = a;
    }
    friend void max(XYZ, ABC);
};

void max(XYZ t1, ABC t2)
{
    if (t1.x > t2.y)
        cout << t1.x;
    else
        cout << t2.y;
}

int main()
{
    ABC a1;
```

```
    XYZ x1;
    a1.set_data(20);
    x1.set_data(35);
    max(a1,x1);
    return 0;
}
```

Output:
35

Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.
We can declare a friend class in C++ by using the **friend** keyword.

Syntax:
friend class class_name; // declared in the base class

Example: *Program to demonstrate friend class*

```
#include <iostream>
using namespace std;
class Demo {
     private:
          int a;
     protected:
           int b;
     public:
    Demo()                                  //default constructor
    {
       a = 100;
       b = 20;
    }
     friend class F;                        // friend class declaration
};
class F {
public:
    void display(Demo d)
    {
       cout << "The value of Private Variable = " << d.a << endl;
       cout << "The value of Protected Variable = " << d.b<<endl;
```

```
    }
};
int main()
{
    Demo d1;
    F f1;
    f1.display(d1);
    return 0;
}
```

Output:
The value of Private Variable = 100
The value of Protected Variable = 20

Operator Overloading in C++ Using Friend Function

1. The Friend function in C++ using operator overloading offers better flexibility to the class.
2. The Friend functions are not a member of the class and hence they do not have 'this' pointer.
3. When we overload a unary operator, we need to pass one argument.
4. When we overload a binary operator, we need to pass two arguments.
5. The friend function in C++ can access the private data members of a class directly.
6. An overloaded operator friend could be declared in either the private or public section of a class.
7. When redefining the meaning of an operator by operator overloading the friend function, we cannot change its basic meaning.

Syntax

```
friend return_type operator operator_symbol (parameter 1, parameter2)
{
   // function definition
}
```

Friend function cannot overload the following operators =, ( ), [ ], ->

Unary Operators Overloading using Friend Functions

Example: Program to demonstrate unary operator "++" overloading using friend function

```
#include <iostream>
using namespace std;
class funO {
private:
        int x;
public:
        funO(int x) {this->x=x;}
        friend void operator ++(funO &);
        void display() {
                cout << "The Count is: " << x;
                }
};
void operator ++(funO &op)
 {
        ++op.x;
}
int main() {
        funO ob(20);
        ++ob;
        ob.display();
        return 0;
}
```

Output:
The Count is: 21

Binary Operators Overloading using Friend Functions

Example: *Program to demonstrate binary operator "+" overloading to add objects of two different classes*

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

class B;
class A
{
   int x;
   public:
    A(int x)
    {
       this->x=x;
```

```
    }
    void friend operator +(A ,B );
};
class B
{
   int x;
   public:
     B(int x)
    {
         this->x=x;
    }
    void friend operator +(A,B);
};
 void operator +(A a, B b)
 {
   cout<<a.x+b.x;
 }
int main()
{
  A ob1(20);
  B ob2(30);
  ob1+ob2;
  return 0;
}
```

Output:
50

Example: *Program to demonstrate binary operator "+" overloading to add two objects of same class. To add to complex numbers.*

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

class ComplexAdd {
private:
        int real, imag;
public:
      ComplexAdd(int r = 0, int i = 0) {
              this->real = r;
              this->imag = i;
        }
```

```
        friend ComplexAdd operator +(ComplexAdd, ComplexAdd );
        void print() {
                cout << real << " + i" << imag << endl;
        }
};
ComplexAdd operator +(ComplexAdd ob1, ComplexAdd ob2) {
                ComplexAdd result(0,0) ;
                result.real = ob1.real + ob2.real;
                result.imag = ob1.imag + ob2.imag;
                return result;
        }

int main()
{
        ComplexAdd c1(10,5);
         ComplexAdd c2(4,2);
        ComplexAdd c3(0,0);
         c3=c1+c2;
        c3.print();
}
```

Output:
14+i 7

"<<" operator overloading using friend function

Overloading stream insertion (<>) operators in C++

- In C++, stream insertion operator "<<" is used for output and extraction operator ">>" is used for input.
- **cout** is an object of **ostream class** and **cin** is an object of **istream class**
- These operators must be overloaded as a global function and if we want to allow them to access private data members of the class, we must make them friend.

Example: *Program to demonstrate "<<" operator overloading*

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

class Fraction {
```

```
private:
        int numerator;
        int denominator;
public:
        // constructor
        Fraction(int x = 0, int y = 1)
        {
                numerator = x;
                denominator = y;
        }
// Operator Overloading performed and friend function used because of two
//classes
        friend istream& operator >>(istream& in, Fraction& c)
        {
                in >> c.numerator >> c.denominator;
                return in;
        }

        friend ostream& operator<<(ostream& out, Fraction& c)
        {
                out << c.numerator << "/" << c.denominator;
                return out;
        }
};

int main()
{
        Fraction ob;
        cout << "Enter a numerator and denominator of the Fraction: ";
        cin >> ob;
        cout << "Fraction is: ";
        cout << ob;
        return 0;
}
```

Output:

Enter a numerator and denominator of the Fraction: 5 10
Fraction is: 5/10

Example: *Program to demonstrate "<<" operator overloading to read and display a Complex number*

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

class Complex
{
  private:
    int real,imag;
  public:
    Complex(int r = 0, int i =0)
    {
          real = r;
          imag = i;
    }
    friend ostream& operator << (ostream &out, Complex &);
    friend istream& operator >> (istream &in, Complex &);
};

istream& operator >> (istream &in, Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imaginary Part ";
    in >> c.imag;
    return in;
}

ostream& operator << (ostream &out, Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}

int main()
{
   Complex c1;
   cin >> c1;
   cout << "The complex object is ";
   cout << c1;
```

```
  return 0;
}
```

Output:

```
Enter Real Part 5
Enter Imaginary Part 10
The complex object is 5 + i 10
```

We can effectively use Operator overloading to increase the usability and expressiveness of C++ classes.