Object Oriented Programming with C++(BCS306B)

Module – 3

Chapter 1 :

OPERATOR OVERLOADING :

SYLLABUS:

Operator Overloading: Creating a Member Operator Function, Operator Overloading Using a Friend Function, Overloading new and delete. Inheritance: Base-Class Access Control, Inheritance and Protected Members, Inheriting Multiple Base Classes , Constructors, Destructors and Inheritance, Granting Access, Virtual Base Classes.

C++ 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.

```
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.

```
// 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.

3.1 Creating a Member Operator Function

- Operator overloading is the process of making an operator exhibit different behaviors in different instances.

Ways of operator overloading

1) Operator overloading of member functions.
2) Operator overloading of non-member or friend functions.

1. Operator overloading of member function :

- Member functions are operators and functions declared as members of a certain class. They don't include operators and functions declared with the friend keyword.
- If you write an operator function as a member function, it gains access to all of the member variables and functions of that class.

- Using operator overloading in C++, we can specify more than one meaning for an operator in one scope.
- The purpose of operator overloading is to provide a special meaning of an operator for a user-defined data type.

```
class className {
   ... .. ...
   public
     returnType operator symbol (arguments) {
        ... .. ...
     }
   ... .. ...
};
```

Here,

returnType is the return type of the function.

the **operator** is a keyword.

the **symbol** is the operator that we want to overload. Like: +, <, -, ++, etc.

arguments are the arguments passed to the function.

Complete Example to add two Complex Numbers in C++:

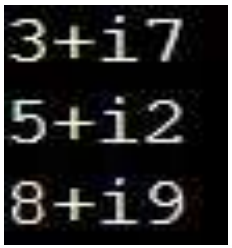
```
#include<iostream>
using namespace std;

class Complex
{
    private:
        int real;
        int img;
    public:
        Complex (int r = 0, int i = 0)
        {
            real = r;
            img = i;
        }
        Complex add (Complex x)
        {
            Complex temp;
            temp.real = real + x.real;
            temp.img = img + x.img;
            return temp;
        }
```

```
void Display() {
            cout << real << "+i" << img <<endl;
               }
};

int main()
{
    Complex C1 (3, 7);
    C1.Display();
    Complex C2 (5, 2);
    C2.Display();
    Complex C3;
    C3 = C1.add (C2);  // C2.add(C1);
    C3.Display();
}
```

OUTPUT :



Example to add two Complex Numbers in C++ using Operator Overloading:

```
#include<iostream>
using namespace std;
class Complex
{
    private:
        int real;
        int img;
    public:
        Complex (int r = 0, int i = 0)
        {
            real = r;
            img = i;
        }
        Complex operator + (Complex x)
        {
            Complex temp;
            temp.real = real + x.real;
            temp.img = img + x.img;
            return temp;
        }
```

```
void Display() {
            cout << real << "+i" << img <<endl;
        }
};

int main()
{
    Complex C1 (3, 7);
    C1.Display();
    Complex C2 (5, 2);
    C2.Display();
    Complex C3;
    C3 = C1+C2;
    C3.Display();
}
```

Output:

```
3+i7
5+i2
8+i9
```

- Understanding logic is the most important thing. So, we have finished it. Now let us see how to make it as operator overloading. Now we want to convert add function into an operator.
- So, instead of writing **C3 = C2.add(C1),** we want to write **C3 = C2 + C1;** So, for writing like this, we have to modify the function signature as follows:

```
Complex operator + (Complex x)
  {
        …
  }
```

- Here, we just replace the **add** word with **operator +.** Everything inside the function will remain the same as the previous one. With the above changes in place, now the + operator is overloaded for class Complex.
- This is Operator Overloading in C++. So instead of writing dot, you can just write '+' to get the addition of two Complex objects.

3.2 Operator Overloading Using a Friend Function

- A non-member function does not have access to the private data of that class.
- This means that an operator overloading function must be made a friend function if it requires access to the private members of the class.

Syntax:

```
friend return-type operator operator-symbol (Variable 1,
Varibale2)
{
      //Statements;
}
```

- Operator Overloading is the method by which we can change the function of some specific operators to do some different tasks.
- We have also given you an example that if ‘X’ has some money and ‘Y’ also has some money and they wanted to add their money.
- So ‘X’ can add money or ‘Y’ can add money or they can also take help from another person i.e. friends. If their friend is adding the money then they both have to give their money to him as the parameter.
- Then only their friend can add the money. So, the same approach will follow for the friend function.

```
#include <iostream>
using namespace std;
class Complex
{
    private:
        int real;
        int img;
    public:
        Complex (int r = 0, int i = 0)
        {
            real = r;
            img = i;
        }
        void Display ()
        {
            cout << real << "+i" << img;
        }
        friend Complex operator + (Complex c1, Complex c2);
};
```

```
Complex operator + (Complex c1, Complex c2)
{
    Complex temp;
    temp.real = c1.real + c2.real;
    temp.img = c1.img + c2.img;
    return temp;
}

int main ()
{
    Complex C1(5, 3), C2(10, 5), C3;
    C1.Display();
    cout << " + ";
    C2.Display();
    cout << " = ";
    C3 = C1 + C2;
    C3.Display();
}
```

OUTPUT:

5+i3 + 10+i5 = 15+i8

3.3 Overloading new and delete

The new operator allocates memory to a variable. For example,

```
    // declare an int pointer
int* pointVar;
   // dynamically allocate memory using the new keyword
pointVar = new int;
  // assign value to allocated memory
*pointVar = 45;
```

Here, we have dynamically allocated memory for an **int** variable using the new operator.

Notice that we have used the pointer **pointVar** to allocate the memory dynamically

This is because the new operator returns the address of the memory location.

In the case of an array, the new operator returns the address of the first element of the array.

From the example above, we can see that the syntax for using the new operator is

```
pointerVariable = new dataType;
```

delete Operator

Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.

For this, the delete operator is used. It returns the memory to the operating system. This is known as memory deallocation.

The syntax for this operator is

```
delete pointerVariable;
```

The general form of operator overloading new

The general form of overloading the new operator is as follows:

```
return_type * operator new(size_t size)
{
    // Memory allocation
    // ...
}
```

here

•***return_type*** is a type (class) to which the operator function returns a pointer for which memory is allocated by a special (non-standard method);

•***size*** – ***size*** of memory that is allocated for ***return_type*** type.

(**size_t** is essentially an unsigned integer.) The parameter **size** will contain the number of bytes needed to hold the object being allocated.

The general form of operator overload delete

The general form of operator overload delete is as follows:

```
void operator delete(void * pointer)
{
    // freeing the memory pointed to by the pointer pointer
    // ...
}
```

here **pointer** – a pointer to the memory area that was previously allocated by the operator new.

Why overload new and delete?

- You might choose to do this if you want to use some special allocation method.
- For example, you may want allocation routines that automatically begin using a disk file as virtual memory when the heap has been exhausted.

- The new and delete operators can also be overloaded like other operators in C++. New and Delete operators can be overloaded globally or they can be overloaded for specific classes.
- If these operators are overloaded using member function for a class, it means that these operators are overloaded **only for that specific class**.
- If overloading is done outside a class (i.e. it is not a member function of a class), the overloaded 'new' and 'delete' will be called anytime you make use of these operators (within classes or outside classes). This is **global overloading**.

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

class MyClass
{
    int num;
public:
    MyClass()
    {
    }

    MyClass(int a):num(a)
    {

    }
    void display()
    {
        cout<< "num:" << num << endl;
    }
```

```
void * operator new(size_t size)
    {
   cout<< "Overloading new operator with size: " << size
         << endl;
        void * p = ::new MyClass();
        //void * p = malloc(size); will also work fine

        return p;
    }
    void operator delete(void * p)
    {
        cout<< "Overloading delete operator " << endl;
        free(p);
    }
};

int main()
{
    MyClass * p = new MyClass(24);
    p->display();
    delete p;
}
```

size_t is a special unsigned integer type defined in the standard library of C and C++.

The size_t is chosen so that it can store the maximum size of a theoretically possible.

OUTPUT:

```
Overloading new operator with size: 4
num:24
Overloading delete operator
```

Module – 3
Chapter 2 :
INHERITANCE:

3.4 Base-Class Access Control

When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form:

```
class derived-class-name : access base-class-name
{
  // body of class
};
```

- The access status of the base-class members inside the derived class is determined by *access*. The base-class access specifier must be either **public, private,** or **protected**.
- If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier.

- When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class.
- In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.

In C++ inheritance, we can derive a child class from the base class in different access modes. For example,

```
class Base
{
.... ... ....
};

class Derived : public Base
{
.... ... ....
};
```

Notice the keyword **public** in the code

Syntax:-

```
class Base
{
   public:int a;
   protected:int b;
   private:int c;
};
class PublicDerived: public Base {
// a is public
// b is protected
// c is not accessible from PublicDerived
};
class ProtectedDerived: protected Base {
// a is protected
// b is protected
// c is not accessible from ProtectedDerived
};
class PrivateDerived: private Base {
// a is private
// b is private
// c is not accessible from PrivateDerived
}
```

```
#include <iostream>
using namespace std;
class Base
{
  private:
     int pvt = 1;
  protected:
     int prot = 2;
  public:
     int pub = 3;
    // function to access private member
  int getPVT()
    {
    return pvt;
    }
};
```

```
class PublicDerived : public Base
{
  public:// function to access protected member from Base
         int getProt()
           {
         return prot;
           }
};

int main()
{
  PublicDerived object1;
  cout << "Private = " <<  object1.getPVT() << endl;
  cout << "Protected = " <<  object1.getProt() << endl;
  cout << "Public = " <<  object1.pub << endl;
  return 0;
}
```

Output :

Private = 1

Protected = 2

Public = 3

Here, we have derived PublicDerived from Base in **public mode.**

As a result, in PublicDerived:

- prot is inherited as **protected.**
- pub and getPVT() are inherited as **public.**
- pvt is inaccessible since it is **private** in Base.

Since **private** and **protected** members are not accessible, we need to create public functions getPVT() and getProt() to access them:

3.5 Inheritance and Protected Members

Protected mode – If the derived class inherits the properties of the base class in **protected** mode, then both **public** and **protected** members will become **protected** in the derived class.

```
/* Program to show the Protected Inheritance */
#include<iostream>
using namespace std; /* Base Class */
class Base
{
    protected:
            int b=2;
    private:
            int c=3;
    public:
            int a=1;
    // Function to access private data member of the class
    int getPrivate()
    {
        return c;
    }
};
```

```
/* Derived Class is inheriting the Base Class protectedly*/
class Derived: protected Base
{
    public:
// Function to access the public value of the base class
    int getPublic()
    {
        return a;
    }
// Function to access the protected value of the base class
    int getProtected()
    {
        return b;
    }
};
```

```
int main() /* Driver Function */
{
    Derived obj;
    cout<<"Public member for the given class is:
        "<<obj.getPublic()<<endl;
    cout<<"Private members are not accessible in the
         derived class"<<endl;
    cout<<"Protected member for the given class is:
       "<<obj.getProtected()<<endl;
 return 0;
}
```

Output:Public member for the given class is: 1

Private members are not accessible in the derived class

Protected member for the given class is: 2

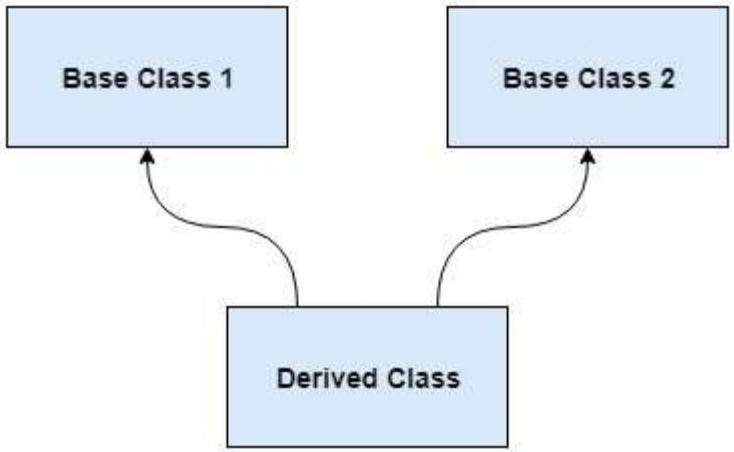
- Here, the private members are not accessible since we are accessing the members of the base class in protected mode.
- We can access the protected and public member of the base class but they are inaccessible directly since we inherited them protected, so we need to create a function for accessing them.

3.6 Inheriting Multiple Base Classes

- When we acquire the features and functionalities of one class to another class, the process is called **Inheritance**.
- In this way, we can reuse, extend or modify all the attributes and behavior of the parent class using the derived class object. It is the most important feature of object-oriented programming that reduces the length of the program.
- A class that inherits all member functions and functionality from another or parent class is called the **derived class**. And the class from which derive class acquires some features is called the **base or parent class**.

Multiple inheritance occurs when a class inherits from more than one base class.

A diagram that demonstrates multiple inheritance is given below .



Multiple Inheritance

In the above diagram, there are two-parent classes:

Base Class 1 and **Base Class 2**, whereas there is only one **Child Class**. The Child Class acquires all features from both Base class 1 and Base class 2. Therefore, we termed the type of Inheritance as Multiple Inheritance.

Syntax of the Multiple Inheritance :

```
class A
 {
   // code of class A
 }
class B
 {
   // code of class B
 }
class C: public A, public B (access modifier class_name)

 {
  // code of the derived class
 }
```

In the above syntax, class A and class B are two base classes, and class C is the child class that inherits some features of the parent classes.

Example 1: Program to use the Multiple Inheritance

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

// create a base class1
class Base_class
{
    // access specifier
    public:
    // It is a member function
    void display()
    {
        cout << " It is the first function of the Base
 class " << endl;
    }
};
```

```
class Base_class2  // create a base class2
{
     public:   // access specifier
    void display2()       // It is a member function
    {
    cout << " It is the second function of the Base class "
 << endl;
    }
};
/* create a child_class to inherit features of Base_class a
nd Base_class2 with access specifier. */
class child_class: public Base_class, public Base_class2
{
    public:       // access specifier
    void display3() // member function of derive class
    {
        cout << " It is the function of the derived class "
 << endl;
    }
};
```

```
int main ()
{
child_class ch;  // create an object for derived class
ch.display();      // call member function of Base_class1
ch.display2();  // call member function of Base_class2
ch.display3();    // call member function of child_class
}
```

Output :

```
It is the first function of the Base class
It is the second function of the Base class
It is the function of the derived class
```

3.7 Constructors

Constructor in C++ is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure.

```
The prototype of the constructor looks like
<class-name> (list-of-parameters);
```

Characteristics of constructor :

- The name of the constructor is same as its class name.
- Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor can not be declared virtual.

Types of constructor

- Default constructor
- Parameterized constructor
- Overloaded constructor
- Constructor with default value
- Copy constructor
- Inline constructor

A constructor in C++ is a special method that is automatically called when an object of a class is created.
A constructor with no parameters is known as a **default constructor**.

To create a constructor,
use the same name as the class, followed by parentheses ():

```
class MyClass {       // The class
  public:             // Access specifier
    MyClass() {       // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;      // Create an object of MyClass (this
will call the constructor)
  return 0;
}
```

Output:
Hello World

```
// Example: defining the constructor within the class

#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student()
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }
```

```
void display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }
};

int main()
{
    student s;  //constructor gets called automatically
                when we create the object of the class
    s.display();
    return 0;

}
```

OutPut:

Enter the RollNo:Enter the Name:Enter the Fee: 2 0

Example: defining the constructor outside the class

```
#include<iostream>
using namespace std;
class student
{
      int rno;
      char name[50];
      double fee;
      public:
      student();
      void display();

};
      student::student()
      {
            cout<<"Enter the RollNo:";
            cin>>rno;
            cout<<"Enter the Name:";
            cin>>name;
            cout<<"Enter the Fee:";
            cin>>fee;
      }
```

```
void student::display()
	{
		cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
	}

int main()
{
	student s;
	s.display();
	return 0;
}
```

Output:

Enter the RollNo: 120
Enter the Name: A
Enter the Fee:10
120 A 10

3.8 Destructors and Inheritance

- The destructor of derived class will be called first then destructor of base class which is mentioned in the derived class declaration is called from last towards first sequence wise.
- When we are using the constructors and destructors in the inheritance, parent class constructors and destructors are accessible to the child class hence when we create an object for the child class, constructors and destructors of both parent and child class get executed.

Order of Inheritance

Order of Constructor Call		Order of Destructor Call	
1. C()	(Class C's Constructor)	1. ~A()	(Class A's Destructor)
2. B()	(Class B's Constructor)	2. ~B()	(Class B's Destructor)
3. A()	(Class A's Constructor)	3. ~C()	(Class C's Destructor)

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

class parent      //parent class
{
    public:
    parent()      //constructor
    {
        cout << "Parent class Constructor\n";
    }

    ~parent()     //destructor
    {
        cout << "Parent class Destructor\n";
    }
};
```

```
class child : public parent   //child class
{
    public:
    child() //constructor
    {
    cout << "Child class Constructor\n";
    }

    ~ child() //destructor
    {
    cout << "Child class Destructor\n";
    }
};
int main()
{
    //automatically executes both child and parent class
    //constructors and destructors because of inheritance
    child c;

    return 0;
}
```

Output :

```
Parent class Constructor

Child class Constructor

Child class Destructor

Parent class Destructor
```

3.9 Granting Access

- When a base class is inherited as **private**, all public and protected members of that class become private members of the derived class.
- However, in certain circumstances, you may want to restore one or more inherited members to their original access specification.
- For example, you might want to grant certain public members of the base class public status in the derived class even though the base class is inherited as **private**.
- In Standard C++, you have two ways to accomplish this. First, you can use a **using** statement, which is the preferred way. The **using** statement is designed primarily to support namespaces.

- The second way to restore an inherited member's access specification is to employ an *access declaration* within the derived class.

An access declaration takes this general form:

```
base-class::member;
```

- The access declaration is put under the appropriate access heading in the derived class' declaration. Notice that no type declaration is required (or, indeed, allowed) in an access declaration.

An access declaration takes this general form:

```
base-class::member;
class base {
       public:    // public in base
       int j;
           };
class derived: private base // Inherit base as private.
 {
  public:    // here is access declaration
      base::j; // make j public again    // ...
 };
```

Because base is inherited as private by derived, the public variable j is made a private variable of derived.

- Because **base** is inherited as **private** by **derived**, the public member **j** is made a private member of **derived.** However, by including

```
base::j;
```

as the access declaration under **derived**'s **public** heading, **j** is restored to its public status.

- You can use an access declaration to restore the access rights of public and protected members. However, you cannot use an access declaration to raise or lower a member's access status.
- For example, a member declared as private in a base class cannot be made public by a derived class.

in C++, While granting access demoting a variable from protected to public is not allowed but it is happening.

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

class base {
                protected: int x;     // x is protected
            };

class derived: private base
{
    public: base::x;
   //demoting from protected to public must not happen
};
int main()
{
    derived d1;
    d1.x=10;
 //protected variable x is being accessed using an object**
    cout<<d1.x<<endl;
}
```

3.10 Virtual Base Classes

- An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited. For example, consider this incorrect program:
- The virtual base class is a concept used in multiple inheritances to prevent ambiguity between multiple instances. For example: suppose we created a class “A” and two classes “B” and “C”, are being derived from class “A”. But once we create a class “D” which is being derived from class “B” and “C” as shown in figure 1.

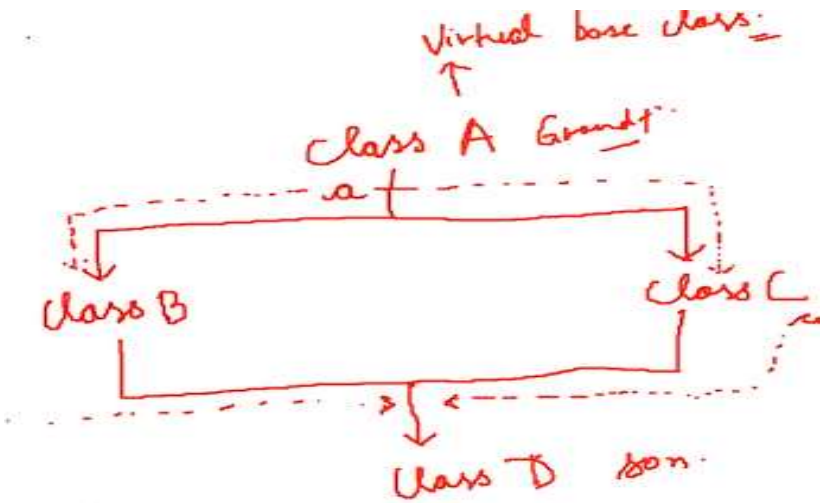


Figure 1: Virtual Base Class Example Diagram

As shown in figure 1,

Class “A” is a parent class of two classes “B” and “C”

And both “B” and “C” classes are the parent of class “D”

- The main thing to note here is that the data members and member functions of class “A” will be inherited twice in class “D” because class “B” and “C” are the parent classes of class “D” and they both are being derived from class “A”.
- So when the class “D” will try to access the data member or member function of class “A” it will cause ambiguity for the compiler and the compiler will throw an error.
- To solve this ambiguity we will make class “A” as a virtual base class. To make a virtual base class “virtual” keyword is used.

- When one class is made virtual then only one copy of its data member and member function is passed to the classes inheriting it.
- So in our example when we will make class “A” a virtual class then only one copy of the data member and member function will be passed to the classes “B” and “C” which will be shared between all classes. This will help to solve the ambiguity.

```
using namespace std;
class A {
      public:
            int a;
      A(){
            a = 10;
         }
       };
class B : public virtual A {};
class C : public virtual A {};
class D : public B, public C {};
```

```
int main()
{
//creating class D object
D object;
cout << "a = " << object.a << endl;
return 0;
}
```

Output :

a = 10