Unit-4
Constructor

Constructor is special member function of a class that initializes an instance of its class.
It has the same name as the class name and does not have return value, not even void.
It can have any number of parameters and a class may have any number of overloaded constructors.
Constructors may have any accessibility, public, private or protected.

Example

```
class employee
{
    private:
       //Employee details;
    public:
       employee();   //Constructor
};
```

Constructors can be defined either inside or outside class definition.
Constructors can be defined outside the class definition using class name and scope resolution (::) operator.

Example

```
class employee
{
    private:
      //Employee Details
    public:
      employee();   //Constructor declared
};
employee::employee()   //Constructor definition outside class definition
{
   //Statements;
}
```

Types of Constructors

Following are the types of constructors:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

1. Default Constructor

Default constructor does not have any parameter.
This type of constructor is important for initialization of object members.

Syntax:

```
class_name()
{
   //Constructor Definition
}
```

Example : Demonstrating the Default Constructor

```
#include<iostream>
class Rectangle
{
    public:
      float l,b;
      Rectangle()   //Constructor created
      {
         l=3;
         b=6;
      }
};
int main()
{
    Rectangle rect;
    cout <<"Area of Rectangle : "<<rect.l*rect.b;
}
```

Output:
Area of Rectangle : 18

In the above program, first object rect belonging to class Rectangle is created and then the constructor that initializes its data members.

Example : Demonstrating how the default constructor is called by the compiler

```
#include<iostream.h>
class Rectangle
{
    public: int l=3,b=3;
};
int main()
{
    Rectangle rect;
    cout<<"Area of Rectangle : "<<rect.l*rect.b;
}
```

Output:
Area of Rectangle : 9

In the above program, **Default Constructor is called automatically by the compiler** which initializes the object data members that is **l & b** to default value (3,3). So the **Area of Rectangle is 9.**
If constructor does not explicitly defined then the compiler will provide a default constructor implicitly.

2. Parameterized Constructor

As we know,default constructor does not have any parameters, but if you need to add parameters to the constructor, you can add to this and this constructor is called as **Parameterized constructor.**
Parameterized Constructor is defined with the parameters which provide different values to data members of different objects by passing the values as an argument.

Example : Demonstrating the Parameterized Constructor

```
#include<iostream.h>
class Square
{
    public:
       float side;
       Square(float s)  //Parameterized Constructor
       {
           side = s;
       }
};
int main()
{
     Square sq(5);
     cout <<"Area of Square : "<<sq.side*sq.side<<endl;
}
```

Output:
Area of Square : 25

3. Copy Constructor

Copy constructor is a special type of constructor which takes an object as an argument.
It is a member function which initializes an object using another object of the same class.
Copy constructor creates a new object which is exact copy of the existing constructor, hence it is called as **Copy Constructor.**
This type of constructor is used to copy values of data members of one object into other object.
It is used to create a copy of an already existing object if a class type.

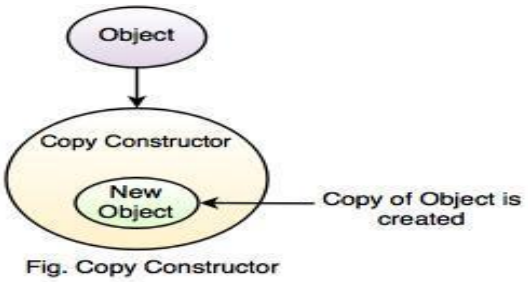

Fig. Copy Constructor

In copy constructor, for copying the object values, both objects must belong to the same class.

Syntax:

```
class_name (const class_name &obj)
{
   //body of constructor;
}
```

Example : Program demonstrating the Copy Constructor

```
#include<iostream.h>
class CopyConst
{
   private:
      int a, b;
   public:
      CopyConst(int a1, int b1)
      {
         a = a1;
         b = b1;
      }
      CopyConst(const CopyConst &obj2) // Copy constructor
      {
         a = obj2.a;
         b = obj2.b;
      }
      int getA()
      {
         return a;
      }
      int getB()
      {
         return b;
      }
};
int main()
{
   CopyConst obj1(10, 20);     // Normal constructor is called
```

```
    cout<<"Normal Constructor"<<endl;
    cout << "Value of a and b" <<"\n a is : "<< obj1.getA() <<"\n b is : "<<
obj1.getB()<<endl;    //Access values assigned by constructors
    cout<<"\n Copy Constructor";
    CopyConst obj2 = obj1;        // Copy constructor is called
    cout << "\n Value of a and b" <<"\n a is : "<< obj2.getA() <<"\n b is : "<<
obj2.getB();
    return 0;
}
```

Output:

```
Normal Constructor
Value of a and b
a is : 10
b is : 20

Copy Constructor
Value of a and b
a is : 10
b is : 20
```

Constructor Overloading

Constructors can be overloaded just like the member functions.
It is used to increase the flexibility of a class by having number of constructor for a single class.

Example : Program demonstrating the Constructor Overloading

```
#include <iostream.h>
class OverloadConst
{
    public:
        int a;
        int b;
        OverloadConst()
        {
            a = b = 0;
        }
        OverloadConst(int c)
        {
            a = b = c;
        }
        OverloadConst(int a1, int b1)
```

```
        {
            a = a1;
            b = b1;
        }
};
int main()
{
    OverloadConst obj;
    OverloadConst obj1(10);
    OverloadConst obj2(20, 30);
    cout << "OverloadConst obj's a & b value : " <<
    obj.a << " , "<< obj.b << "\n";
    cout << "OverloadConst obj1's a & b value : "<<
   obj1.a << " ,"<< obj1.b << "\n";
   cout << "OverloadConst obj2's  a & b value : "<<
   obj2.a << " , "<< obj2.b << "\n";
   return 0;
}
```

Output:

```
OverloadConst obj's a & b value : 0 , 0
OverloadConst obj1's a & b value : 10 ,10
OverloadConst obj2's a & b value : 20 , 30
```

In the above example, the constructor **OverloadConst** is overloaded thrice with different initialized values.

Destructor

Destructor is a type of special member function of a class.
It is used to destroy the memory allocated by the constructor.
It has the same name as the class prefixed with a **tilde (~) sign.**
Destructor does not take any arguments and cannot return or accept any value.
It is called automatically by the compiler when the object goes out of scope.
Compiler calls the destructor implicitly when the program execution is exited.

Syntax:

```
class class_name
{
   public:
     ~class_name();
};
```

Example : Demonstrating the Destructor

```
#include <iostream.h>
int count=0;
class show
{
    public:
       show()
       {
          count++;
          cout << "Create Object : " << count<<endl;
       }
       ~show()
       {
          cout << "Destroyed Object : " << count<<endl;
          count--;
       }
};

int main()
{
    cout << "Main Objects: a,b,c\n";
    show a,b,c;
    {
        cout << "\n New object: d\n";
        show d;
    }
    cout << "\n Destroy All objects: a,b,c\n";
    return 0;
}
```

Output:

Main Objects: a,b,c
Create Object : 1
Create Object : 2
Create Object : 3

New object: d
Create Object : 4
Destroyed Object : 4

Destroy All objects: a,b,c
Destroyed Object : 3
Destroyed Object : 2
Destroyed Object : 1

In the above program, constructors **show()** and destructor **~show()** is used.

First three objects **a,b,c** are created and fourth object **d** is created inside **"{}".** The fourth object **d** is destroyed implicitly when the code execution goes out of scope defined by curly braces **({}).** And then, all the existing objects **a,b,c** are destroyed.

Unit-4
Inheritance

The mechanism of deriving a class from another class is known as **Inheritance.**
Inheritance is the most importance concept of object oriented programming.
It allows us to define a class in terms of another class, which helps to create and maintain an application.

The main advantage of Inheritance is, it provides an opportunity to reuse the code functionality and fast implementation time.
The members of the class can be Public, Private or Protected.

Syntax:

class DerivedClass : AccessSpecifier BaseClass
The default access specifier is **Private.**
Inheritance helps user to create a new class (derived class) from a existing class (base class).
Derived class inherits all the features from a Base class including additional feature of its own.

Access Control

Accessibility	Private	Public	Protected
From own class	Yes	Yes	Yes
From derived class	No	Yes	Yes

Outside derived class	No	Yes	No

Example: Program demonstrating implementation of Inheritance

```
#include <iostream.h>
class Shape
{
  protected:
      float width, height;
  public:
      void set_data (float w, float h)
      {
            width = w;
            height = h;
      }
};

class Rectangle: public Shape
{
   public:
       float area ()
       {
             return (width * height);
       }
};

class Triangle: public Shape
{
   public:
      float area ()
      {
             return (width * height / 2);
      }
};

int main ()
{
      Rectangle r;
      Triangle t;
      r.set_data (4,4);
      t.set_data (3,7);
      cout << "Area of Rectangle : "<<r.area() << endl;
      cout << "Area of Triangle : "<<t.area() << endl;
      return 0;
}
```

Output:

Area of Rectangle: 16

Area of Triangle: 10.5

In the above example, **class Shape** is a **base class** and
classes **Rectangle** and **Triangle** are the **derived class.** The derived class appears with the declaration of class followed by a colon(:), access specifier public and the name of the base class from which it is derived.

The **Rectangle** and **Triangle** are derived from **Shape class,** all the data members and member functions of the base **class Shape** can be accessible from derived class.

Types of Inheritance

Following are the types of Inheritance.

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Single Inheritance

In Single Inheritance, one class is derived from another class.
It represents a form of inheritance where there is only one base and derived class.

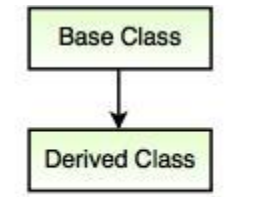

Fig. Single Inheritance

Example: Program demonstrating Single Inheritance

```
#include<iostream.h>
class Student
{
      public:
      int rollno;
      string sname;

      void setdata()
      {
            rollno = 1;
            sname = "ABC";
```

```
        }
};

class Marks : public Student
{
  public:
        int m1, m2, m3, total;
        float per;
        void setmarks()
        {
                setdata();
                m1 = 60;
                m2 = 65;
                m3 = 70;
                total = m1 + m2 + m3;
                per = total / 3;
        }
        void show()
        {
                setmarks();
                cout<<"Student Information"<<endl;
                cout<<"Student Roll No :  "<<rollno<<endl;
                cout<<"Student Name :  "<<sname<<endl;
                cout<<"-----------------------"<<endl;
                cout<<"Student Marks"<<endl;
                cout<<"Marks1 :  "<<m1<<endl;
                cout<<"Marks2 :  "<<m2<<endl;
                cout<<"Marks3 :  "<<m3<<endl;
                cout<<"Total :  "<<total<<endl;
                cout<<"Percentage :  "<<per<<endl;
        }
};

int main()
{
        Marks mar;
        mar.setmarks();
        mar.show();
        return 0;
}
```

Output:

```
Student Information
Student Roll No : 1
Student Name : ABC
----------------------
Student Marks
Marks1 : 60
Marks2 : 65
Marks3 : 70
```

Total : 195
Percentage : 65

In the above example, the **base class Student** has two public member variables namely **rollno** and **sname.** These members are kept public, so that they can accessible from the derived class function to display marks, total and percentage.

The **class Marks** is derived from the **class Student**:
class Marks : public Student

This concept of gaining access to the members of the base class is called as **code reusability.**

The main() function has a first creation of the object of the **class Marks.** The **object mar** is created of **class Marks.**

2. Multiple Inheritance

In Multiple Inheritance, one class is derived from multiple classes.

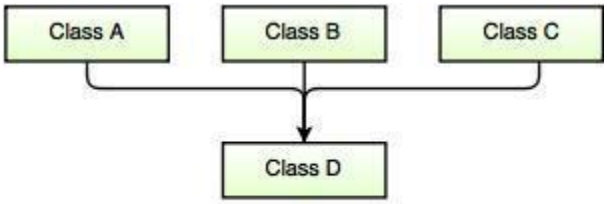

Fig. Multiple Inheritance

In the above figure, Class D is derived from class A, Class B and Class C.

We need to use following **syntax** for deriving a class from multiple classes.

class derived_class_name : access_specifier base_classname1, access_specifier base_classname2, access_specifier base_classname3

Example: Program demonstrating Multiple Inheritance

```
#include<iostream.h>
class A
{
  public:
    A()
    {
        cout << "Class A" << endl;
    }
};

class B
```

```
{
  public:
    B()
    {
        cout << "Class B" << endl;
    }
};

class C
{
  public:
    C()
    {
        cout<<"Class C"<<endl;
    }
};

class D: public C, public B, public A  // Note the order. Class C, Class B and then Class A
{
  public:
    D()
    {
        cout << "Class D" << endl;
    }
};

int main()
{
    D d;
    return 0;
}
```

Output:
Class C
Class B
Class A
Class D

In the above example, **Class C** is called before **Class B** and **Class A**. **Class D** is a derived class from **Class C, Class B** and **Class A.** The constructors of inherited classes are called in the same order in which they are inherited.

In this case, derived class allows to access members of both the base classes from which it is inherited. **Class D** is derived from multiple classes.

3. Multilevel Inheritance

In Multilevel inheritance, one class is derived from a class which is also derived from another class.

It represents a type of inheritance when a derived class is a base class for another class.

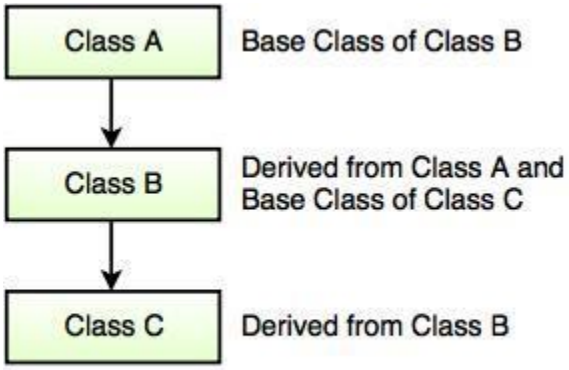

Fig. Multilevel Inheritance

Example: Program demonstrating Multilevel Inheritance

```
#include <iostream.h>
class A
{
   public:
     void show()
     {
        cout<<"Class A - Base Class";
     }
};

class B : public A
{

};

class C : public B
{

};

int main()
{
   C c;
   c.show();
   return 0;
}
```

Output:
Class A - Base Class

In the above program, **Class B** is derived from **Class A** and **Class C** is derived from **Class B. Object c** is created of **Class C** in **main()** function. When the **show()** function is called, **Class A** is executed **show()** because there is

no **show()** function **Class C** and **Class B.** At the time of execution the program first looks for **show()** function in **Class C**, but cannot find it. Then looks in **Class B** because **Class C** is derived from **Class B** and again cannot find it. Finally looks for **show()** function in **Class A** and executes it and displays the output.

If the **show()** function is declared in **Class C** too then it would have override **show()** function in **Class A** because of member function overriding.

4. Hierarchical Inheritance

In Hierarchical Inheritance, there are multiple classes derived from one class.
In this case, multiple derived classes allow to access the members of one base class.

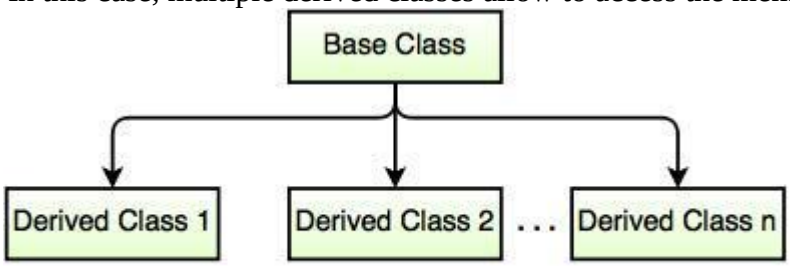

Fig. Hierarchical Inheritance

Syntax:

```
class base_class
{
  . . .
};
class first_derived_class : access_specifier base_class
{
  . . .
};
class second_derived_class : access_specifier base_class
{
  . . .
};
class third_derived_class : access_specifier base_class
{
  . . .
};
```

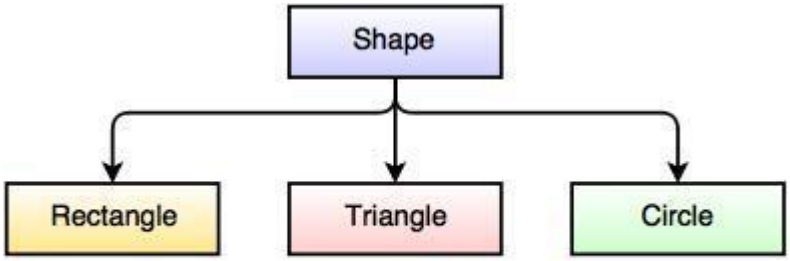

Fig. Hierarchical Inheritance with Class Shape

Example: Program demonstrating Hierarchical Inheritance

```
#include <iostream.h>
class Shape
{
   public:
     Shape()
     {
        cout<<"Base Class - Shape"<<endl;
     }
};

class Rectangle : public Shape
{
       public:
       Rectangle()
       {
              cout<<"Derived Class - Rectangle"<<endl;
       }
};

class Triangle : public Shape
{
       public:
       Triangle()
       {
              cout<<"Derived Class - Triangle"<<endl;
        }
};

class Circle : public Shape
{
       public:
```

```
    Circle()
    {
        cout<<"Derived Class - Circle"<<endl;
    }
};

int main()
{
    Rectangle r;
    cout<<"----------------------------"<<endl;
    Triangle t;
    cout<<"----------------------------"<<endl;
    Circle c;
    return 0;
}
```

Output:

```
Base Class - Shape
Derived Class - Rectangle
--------------------------
Base Class - Shape
Derived Class - Triangle
--------------------------
Base Class - Shape
Derived Class - Circle
```

5. Hybrid Inheritance

Hybrid inheritance is also known as Virtual Inheritance.
It is a combination of two or more inheritance.

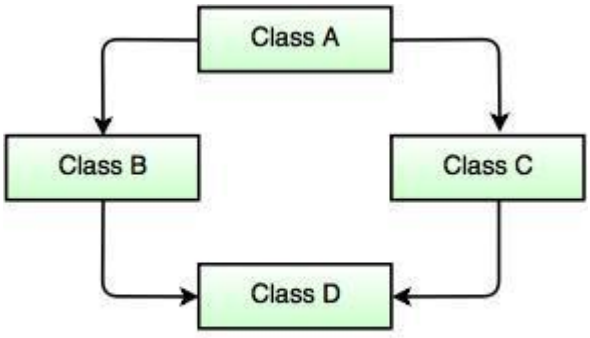

Fig. Hybrid Inheritance

- In hybrid inheritance, when derived class have multiple paths to a base class, a diamond problem occurs. It will result in duplicate inherited members of the base class.

- To avoid this problem easily, use **Virtual Inheritance**. In this case, derived classes should inherit base class by using Virtual Inheritance.

Example: Program demonstrating Hybrid Inheritance

```
#include<iostream.h>
class Student
{
   protected:
        int rollno;
   public:
        void get_rollno(int r)
        {
              rollno=r;
         }
        void show_rollno(void)
        {
              cout<<"Student Result"<<endl;
              cout<<"--------------------------"<<endl;
              cout<<"Roll no : "<<rollno<<"\n";
        }
};

class Test:public virtual Student
{
    protected:
         float mark1, mark2;
    public:
         void get_marks(float m1,float m2)
         {
               mark1 = m1;
               mark2 = m2;
         }
         void show_marks()
         {
               cout<<"Marks obtained:"<<endl;
               cout<<"Mark1 = "<<mark1<<"\n"<<"Mark2 = "<<mark2<<"\n";
         }
};
```

```
class Sports : public virtual Student
{
    protected:
        float score;
    public:
        void get_score(float s)
        {
            score = s;
        }
        void show_score(void)
        {
            cout<<"Sport Scores = "<<score<<"\n";
        }
};

class Result: public Test, public Sports
{
        float total;
    public:
        void show(void);
};

void Result :: show(void)
{
    total = mark1 + mark2 + score;
    show_rollno();
    show_marks();
    show_score();
    cout<<"----------------------------"<<endl;
    cout<<"Total Score = "<<total<<"\n";
}

int main()
{
    Result re;
    re.get_rollno(101);
    re.get_marks(60,65);
    re.get_score(6.0);
    re.show();
    return 0;
}
```

Output:

Student Result

```
---------------------------
Roll no : 101
Marks obtained:
Mark1 = 60
Mark2 = 65
Sport Scores = 6
---------------------------
Total Score = 131
```

In the above program, **base class Student** is created and another class called **Test** and **Sports** are inherited from the main class. **Test class** and **Sports class** inherited by another class called **Result** to calculate the marks.

Unit-4
Operator Overloading

Operator overloading is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.
The main purpose of operator overloading is to perform operation on user defined data type.
For eg. The '+' operator can be overloaded to perform addition on various data types.

Operator overloading is used by the programmer to make a program clearer.
It is an important concept in C++.

Syntax:

```
Return_type Classname :: Operator OperatorSymbol (Argument_List)
{
    //Statements;
}
```

The **operator** keyword is used for overloading the operators.

There are a few operators which cannot be overloaded are follows,

i. Scope resolution operator (::)
ii. sizeof
iii. member selector (.)
iv. member pointer selector (*)
v. ternary operator (? :)

There are some restrictions considered while implementing the operator overloading,

1. The number of operands cannot be changed. Unary operator remains unary, binary remains binary etc.
2. Only existing operators can be overloaded.
3. The precedence and associativity of an operator cannot be changed.
4. Cannot redefine the meaning of a procedure.

• Unary Operator Overloading

Unary operator works with one operand and therefore the user defined data types, operand becomes the caller and hence no arguments are required.

Example : Program demonstrating the Unary Increment & Decrement Operator Overloading

```
#include<iostream.h>
//Increment and Decrement overloading

class IncreDecre
{
    private:
        int cnt ;
    public:
        IncreDecre() //Default constructor
        {
            cnt = 0 ;
        }
        IncreDecre(int C)  // Constructor with Argument
        {
            cnt = C ;
        }
        IncreDecre operator ++ ()     // Operator Function Definition for prefix
        {
            return IncreDecre(++cnt);
        }
        IncreDecre operator ++ (int)       // Operator Function Definition with dummy argument for postfix
        {
            return IncreDecre(cnt++);
        }
        IncreDecre operator -- ()       // Operator Function Definition for prefix
        {
            return IncreDecre(--cnt);
        }
        IncreDecre operator -- (int)       // Operator Function Definition with dummy argument for postfix
        {
            return IncreDecre(cnt--);
        }
        void show()
```

```
        {
            cout << cnt << endl ;
        }
};

int main()
{
    IncreDecre a, b(5), c, d, e(2), f(5);
    cout<<"Unary Increment Operator : "<<endl;
    cout << "Before using the operator ++()\n";
    cout << "a = ";
    a.show();
    cout << "b = ";
    b.show();

    ++a;
    b++;

    cout << "After using the operator ++()\n";
    cout << "a = ";
    a.show();
    cout << "b = ";
    b.show();

    c = ++a;
    d = b++;

    cout << "Result prefix (on a) and postfix (on b)\n";
    cout << "c = ";
    c.show();
    cout << "d = ";
    d.show();
    cout<<"\n Unary Decrement Operator : "<<endl;
    cout << "Before using the operator --()\n";
    cout << "e = ";
    e.show();
    cout << "f = ";
    f.show();

    --e;
    f--;

    cout << "After using the operator --()\n";
    cout << "e = ";
    e.show();
   cout << "f = ";
   f.show();

   c = --e;
   d = f--;

   cout << "Result prefix (on e) and postfix (on f)\n";
   cout << "c = ";
   c.show();
```

```
    cout << "d = ";
    d.show();
    return 0;
}
```

Output:

```
Unary Increment Operator :
Before using the operator ++()
a = 0
b = 5
After using the operator ++()
a = 1
b = 6
Result prefix (on a) and postfix (on b)
c = 2
d = 6

Unary Decrement Operator :
Before using the operator --()
e = 2
f = 5
After using the operator --()
e = 1
f = 4
Result prefix (on e) and postfix (on f)
c = 0
d = 4
```

In the above program, **int** is a **dummy argument** to redefine the functions for the unary **increment (++)** and **decrement (– –)** overloaded operators. Remember one thing **int is not an Integer, it is just a dummy argument.** It is a signal to compiler to create the postfix notation of the operator. **Bjarne Stroustrup** has **introduced the concept of dummy argument,** so it becomes function overloading for the operator overloaded functions.

• Binary Operator Overloading

Binary operator works with two operands.
The first operand becomes the operator overloaded function caller and the second is passed as an argument.

Example : Program demonstrating Binary operator overloading

//Arithmetic operation using Binary Operator Overloading

```
#include<iostream.h>
class BinaryArithmetic
{
```

```
    private:
        float num;
    public:
        void getnumber()
        {
            num = 10;
        }
        BinaryArithmetic operator+(BinaryArithmetic &ab)
        {
            BinaryArithmetic x;
            x.num = num + ab.num;
            return x;
        }
        BinaryArithmetic operator-(BinaryArithmetic &ab)
        {
            BinaryArithmetic x;
            x.num = num - ab.num;
            return x;
        }
        BinaryArithmetic operator*(BinaryArithmetic &ab)
        {
            BinaryArithmetic x;
            x.num = num * ab.num;
            return x;
        }
        BinaryArithmetic operator/(BinaryArithmetic &ab)
        {
            BinaryArithmetic x;
            x.num = num/ab.num;
            return x;
        }
        void show()
        {
            cout<<num;
        }
};

int main()
{
    BinaryArithmetic ba1,ba2,ba3;
    ba1.getnumber();
    ba2.getnumber();
    ba3 = ba1 + ba2;
    cout<<"Addition : ";
    ba3.show();
    ba3 = ba1 - ba2;
    cout<<"\n\n Subtraction : ";
    ba3.show();
    ba3 = ba1 * ba2;
    cout<<"\n\n Multiplication : ";
```

```
    ba3.show();
    ba3 = ba1/ba2;
    cout<<"\n\n Division : ";
    ba3.show();
    return 0;
}
```

Output:
Addition : 20
Subtraction : 0
Multiplication : 100
Division : 1

In the above example, overloading function for addition should be declared as, **BinaryArithmetic operator+(BinaryArithmetic &ab);** where **BinaryArithmetic** is a **class name** and **ab** is an **object.**

To call function **operator()** the statement is as follows:

```
BinaryArithmetic operator+(BinaryArithmetic &ab)
{
    BinaryArithmetic x;
    x.num = num+ab.num;
    return x;
}
```

Member function can be called by using class of that object. The called member function is always preceded by the object.

In the above statement, the object **x** invokes the **operator()** function and the object **ab** is used as an argument for the function. The data member **num** is passed directly. While overloading binary operators, the left-hand operand calls the operator function and the right-hand operator is used as an argument.

Binary operator requires one argument and the argument contains value of the object to the right of the operator.