

UNIT-IV**Syllabus**

- **Operator Overloading and Type Conversion**
 - Overloading operators,
 - rules for overloading operators,
 - overloading of various operators,
 - type conversion - basic type to class type, class type to basic type, class type to another class type.
- **Inheritance**
 - Introduction,
 - defining derived classes,
 - forms of inheritance,
 - ambiguity in multiple and multipath inheritance,
 - virtual base class,
 - object slicing,
 - overriding member functions,
 - object composition and delegation,
 - order of execution of constructors and destructors.

INTRODUCTION

An operator is a symbol that tells the compiler to perform specific task. Every operator have their own functionality to work with built-in data types. Class is user-defined data type and compiler doesn't understand, how to use operators with user-defined data types. To use operators with user-defined data types, they need to be overload according to a programmer's requirement.

Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types. An operator can be overloaded by defining a function to it. The function for operator is declared by using the **operator** keyword followed by the operator.

There are two types of operator overloading in C++

- Binary Operator Overloading
- Unary Operator Overloading

RULES FOR OPERATOR OVERLOADING

- i. Only those operators that are predefined in the c++ compiler can be overloaded. New operators cannot be created such as (\$,# , @).
- ii. Overloaded operator must have at least one operand that is of user-defined type.
- iii. We can't change the basic meaning of an operator, i.e., we can't redefine the + to subtract one value from the other value.
- iv. Overloaded operators follow the syntax rules of the original operators. They can't be overridden.
- v. There are some operators that can't be overloaded:

<u>OPERATOR</u>	<u>MEANING</u>
sizeof ()	Size of operator
.	Membership operator
*	Pointer to member operator
::	Scope resolution operator
?:	Ternary (or) conditional operator

- vi. We cannot use friend functions to overload certain operators(=, (),[],->). However, member functions can be used to overload them.
- vii. Unary operators overloaded by means of a member function take no arguments and return no values. But unary operators overloaded by means of a friend functions take one reference argument.
- viii. Binary operators overloaded through a member functions take one argument and those, which are overloaded through a friend function, take 2 arguments.
- ix. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
- x. Binary arithmetic operators such as +, -, *, / must explicitly return a value. They must not attempt to change their own arguments.

OPERATOR OVERLOADING SYNTAX

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function, which describes the task. The general form of an operator function is

```
return_type operator op(arglist)
{
function body
}
```

Where, return_type--> is the type of value returned by the specified operation
op -->is the operator being overloaded

The op preceded by the keyword **operator**. **operator** op is the function name.

Operator function must be either

- i. member function (no argument for unary operator and one argument for binary)
- ii. friend functions (one argument for unary operator and two argument for binary)

Arguments may be passed either by value or by reference.

The process of overloading involves the following steps:

- i. Create a class that defines the data type that is to be used in the overloading operation.
- ii. Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function.
- iii. Define the operator function to implement the required operations.

OVERLOADING UNARY OPERATOR

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

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

Overloading unary minus(-) operator

We know that unary minus operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is

applied to an int or float variable. The unary minus when applied to an object should change the sign of each of its data items.

Program 1: Program to overload unary minus (-) operator using member function of class.

```
#include<iostream>
using namespace std;
class space
{
int x;
int y;
int z;
public:
void getdata(int a, int b, int c);
void display(void);
void operator-(); //overloaded unary minus
};
void space :: getdata(int a, int b, int c)
{
x=a;
y=b;
z=c;
}
void space :: display(void)
{
cout<<x<<"  ";
cout<<y<<"  ";
cout<<z<<" \n";
}
void space :: operator-()
{
x=-x;
y=-y;
z=-z;
}

int main()
{
space S;
S.getdata(10,-20,30);
cout<<"S :    ";
S.display();
-S;
cout<<"S :    ";
S.display();
return 0;
}
```

Output:

```
S :    10    -20    30
S :    -10    20    -30
```

In the above program, operator function is a member function of the same class, it can directly access the members of the object which activated it.

Program 2: Program to overload unary minus (-) operator using friend function.

```
#include<iostream>
using namespace std;
class space
{
int x;
int y;
int z;
public:
void getdata(int a, int b, int c);
void display(void);
friend void operator-(space &s); //overloaded unary minus
};
void space :: getdata(int a, int b, int c)
{
x=a;
y=b;
z=c;
}
void space :: display(void)
{
cout<<x<<" ";
cout<<y<<" ";
cout<<z<<" \n";
}
void operator-(space &s)
{
s.x=-s.x;
s.y=-s.y;
s.z=-s.z;
}

int main()
{
space s;
s.getdata(10,-20,30);
cout<<"s : ";
s.display();
-s;
cout<<"s : ";
s.display();
return 0;
}
```

Output:

```
s : 10 -20 30
s : -10 20 -30
```

In the above program, the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-(). Therefore, the changes made inside the operator function will not reflect in the called object.

Program 3: Program to overload unary operators that is increment(++) and decrement(--) using member function of class.

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

class IncreDecre
{
    int a, b;
public:
    void accept()
    {
        cout<<"\n Enter Two Numbers : \n";
        cout<<" ";
        cin>>a;
        cout<<" ";
        cin>>b;
    }
    void operator--() //Overload Unary Decrement
    {
        a--;
        b--;
    }
    void operator++() //Overload Unary Increment
    {
        a++;
        b++;
    }
    void display()
    {
        cout<<"\n A : "<<a;
        cout<<"\n B : "<<b;
    }
};

int main()
{
    IncreDecre id;
    id.accept();
    --id;
    cout<<"\n After Decrementing : ";
    id.display();
    ++id;
    ++id;
    cout<<"\n\n After Incrementing : ";
    id.display();
    return 0;
}
```

Output:

```

Enter Two Numbers :
4 5

After Decrementing :
A : 3
B : 4

After Incrementing :
A : 5
B : 6

```

Program 4: Program for unary logical NOT (!) operator overloading using member function of class.

```

#include<iostream>
using namespace std;

class Base
{
    private:
        int a;

    public:
        void read(int x)
        {
            a=x;
        }

        void display(void)
        {
            cout << "value of a is: " << a;
        }

        void operator ! (void)
        {
            a=!a;
        }
};

int main()
{
    Base obj;
    obj.read(10);
    cout << "Before calling Operator Overloading:";
    obj.display();
    cout << endl;
    !obj;
    cout << "After calling Operator Overloading:";
    obj.display();
    cout << endl;
    return 0;
}

```

Output:

```

Before calling Operator Overloading:value of a is: 10
After calling Operator Overloading:value of a is: 0

```

OVERLOADING BINARY OPERATORS

Binary operator is an operator that takes two operand(variable). Binary operator overloading is similar to unary operator overloading except that a binary operator overloading requires an additional parameter.

Binary Operators

- Arithmetic operators (+, -, *, /, %)
- Arithmetic assignment operators (+=, -=, *=, /=, %=)
- Relational operators (>, <, >=, <=, !=, ==)

Program 5: Program to overload '+' operator to add two objects using member function of class.

```
#include<iostream>
using namespace std;
class Test
{
int a;
public:

void getdata()
{
cout<<"enter any number = ";
cin>>a;
}

void display()
{
cout<<"sum="<<a<<endl;
}

Test operator+(Test t1)
{
t1.a=a+t1.a;
return t1;
}
};

int main()
{
Test t1,t2,t3;
t1.getdata();
t2.getdata();
t3=t1+t2;
t3.display();
return 0;
}
```

Output:

```
enter any number = 7
enter any number = 7
sum=14
```

Program 6: Program to overload '+' operator to concatenate two strings using member function of class.


```

#include<iostream>
#include<string.h>
using namespace std;
class test
{
    char str[20];
public:
    void read()
    {
        cout<<"enter the string = ";
        cin>>str;
    }
    void show()
    {
        cout<<"String="<<str<<endl;
    }
    test operator +(test t2)
    {
        test t3;
        strcpy(t3.str, str);
        strcat(t3.str, " ");
        strcat(t3.str, t2.str);
        return t3;
    }
};

int main()
{
    test t1,t2,t3;
    t1.read();
    t2.read();
    t3=t1+t2;
    t3.show();
}

```

Output:

```

enter the string = minu
enter the string = kumari
String=minu  kumari

```

Program 7: Program to overload '+' operator to add two complex number using member function of class .

```

#include <iostream>
using namespace std;
class complex
{
    private:
        float x;
        float y;
    public:
        complex() { }
        complex(float real, float imag)
        {

```

```

        x= real;
        y=imag;
    }

    complex operator + (complex c)
    {
        complex temp;
        temp.x = x +c.x;
        temp.y = y + c.y;
        return temp;
    }
    void display()
    {
        cout<< x << "+" << y << "i"<<endl;
    }
};

int main()
{
    complex c1, c2,c3;
    c1=complex(2.5,3.5); //invokes constructor 1
    c2=complex(1.6,2.7); //invokes constructor 2
    c3=c1+c2; //invokes operator+() function
    cout<<"c1 = "; c1.display();
    cout<<"c2 = "; c2.display();
    cout<<"c3 = "; c3.display();
    return 0;
}

```

Output:

```

c1 = 2.5+3.5i
c2 = 1.6+2.7i
c3 = 4.1+6.2i

```

Features of the above operator function:

- i. It receives only one complex type argument explicitly.
- ii. It returns a complex type value.
- iii. It is a member function of complex.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
c3=c1+c2;
```

We know that a member function can be invoked only by an object of the same class. Here, the object c1 takes the responsibility of invoking the function and c2 plays the role of an argument that is passed to the function. The above invocation statement is equivalent to

```
c3=c1.operator +(c2);
```

Therefore, in the operator+() function, the data members of c1 are accessed directly and the data members of c2 are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
temp.x= x+ c.x;
```

c.x refers to the object c2 and x refers to the object **c1**.

temp.x is the real part of temp that has been created specially to hold the results of addition of c1 and c2. The function returns the complex temp to be assigned to c3.

As a rule, in overloading of binary operators, the left hand operand is used to invoke the operator function and the right hand operand is passed as an argument.

Program 8: Program to overload greater than, less than and '==' operator to compare two objects using member function of class.

```
#include<iostream>
using namespace std;
class Number
{
int n;
public:

void read()
{
cout<<"enter a number"<<endl;
cin>>n;
}

int operator >(Number x)
{
if(n > x.n)
return 1;
else
return 0;
}

int operator <(Number x)
{
if(n < x.n)
return 1;
else
return 0;
}

int operator ==(Number x)
{
if(n == x.n)
return 1;
else
return 0;
}
```

```
};

int main()
{
    Number n1,n2;
    n1.read();
    n2.read();
    if(n1<n2)
        cout<<"n1 is less than n2"<<endl;
    else
        if(n1>n2)
            cout<<"n1 is greater than n2"<<endl;
        else if(n1==n2)
            cout<<"n1 is euqal to n2"<<endl;
    return 0;
}
```

Output:

```
enter value of t1?
enter value of t2?
objects are not equal
```

Program 9: Program to overload ‘==’ operator to compare two strings using member function of class.

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

class test
{
    public:
        char str[20];
    public:
        void get_string()
        {
            cout<<"\n Enter String : ";
            cin>>str;
        }
        void operator==(test &t2)
        {
            if(strcmp(str,t2.str)==0)
            {
                cout<<"Both Strings are Equal";
            }
            else
            {
                cout<<"Strings are not Equal";
            }
        }
};

int main()
{
    test t1, t2;
    int result;
```

```

    t1.get_string();
    t2.get_string();
    t1==t2;    //Comparing two strings. Overloaded '==' operator
    return 0;
}

```

Output:

```

Enter String      :   minu
Enter String      :   minu
Both Strings are Equal

```

Program 10: Program to overload assignment operator using member function of class.

```

#include<iostream>
using namespace std;
class integer
{
    int x;
    int y;
public:
    integer()
    {}
    integer(int x1, int y1) {
        x = x1;
        y = y1;
    }
    void operator=(integer &m ) {
        x = m.x;
        y= m.y;
    }
    void display() {
        cout << "\n  x is:" << x;
        cout << "\n  y is:" << y;
    }
};

int main()
{
    integer j(10, 20);
    integer i;
    i=j;
    j.display();
    i .display();
    return 0;
}

```

Output:

```

x is:10
y is:20
x is:10
y is:20

```

OVERLOADING BINARY OPERATORS USING FRIENDS

Friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitly passed to it, while a member function requires only one.

Program 11: Program to overload '+' operator to add two complex number using friend function.

```
#include<iostream>
using namespace std;
class complex
{
int real,imag;
public:
void set()
{
cout<<"enter real & imag values";
cin>>real>>imag;
}
friend complex operator+(complex,complex);
void display()
{
cout<<"the sum is = "<<real<<"  +  " <<imag<<" i";
}
};
complex operator+(complex t1,complex t2)
{
complex temp;
temp.real=t1.real+t2.real;
temp.imag=t1.imag+t2.imag;
return(temp);
}

int main()
{
complex t1,t2;
t1.set();
t2.set();
t1=t1+t2;
t1.display();
return(0);
}
```

Output:

```
enter real & imag values2 2
enter real & imag values2 2
the sum is = 4 + 4 i
```

Overloading insertion and extraction operators

In C++, stream insertion operator "<<" is used for output and extraction operator ">>" is used for input. We must know following things before we start overloading these operators.

- 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 class, we must make them friend.

Program 12: Program to overload insertion (<<) and extraction(>>) operator using friend function.

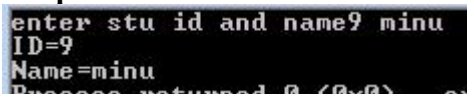
```
#include<iostream>
using namespace std;
class stu
{
    int id;
    char name[10];
public:
    friend void operator>>(istream &in, stu &s);
    friend void operator<<(ostream &out, stu &s);
};

void operator>>(istream &in, stu &s)
{
    cout<<"enter stu id and name";
    in>>s.id>>s.name;
}

void operator<<(ostream &out, stu &s)
{
    out<<"ID="<<s.id<<endl;
    out<<"Name="<<s.name;
}

int main()
{
    stu s;
    cin>>s;
    cout<<s;
    return 0;
}
```

Output:



```
enter stu id and name? minu
ID=9
Name=minu
Process returned 0 (0x0)   0 seconds
```

TYPE CONVERSIONS

The process of converting one predefined type into another is called as **type conversion**. When constants and variables of different types are mixed in an expression, a type conversion will occur. This is so for basic data types.

Therefore, the types of right side and left side of an assignment should be compatible so that type conversion can take place. The compatible data types are char, int, float, double. For example, the following statement :

- int m;
- float x=3.14159;
- m=x;

convert x to an integer before its value is assigned to m. thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types.

The compiler is unknown about the user-defined data type and about their conversion to other data types. The programmer should write the routines that convert basic data type to user-defined data type or vice versa. There are three possibilities of data conversion as given below:

- a. Conversion from basic data type to user-defined data type(class type)
- b. Conversion from class type to basic data type
- c. Conversion from one class type to another class type

Conversion from basic to class type

The conversion from basic to class type can be implemented through constructor. In this type, the left hand operand of = sign is always class type and the right-hand operand is always basic type.

Example:

```
complex c1;
int x=5;
c1=x; //error
```

Program 13: Program to implement conversion from primitive type to class type.

```
#include<iostream>
using namespace std;
class complex
{
    private:
        int a,b;
    public:
        complex()
        {
        }
        complex(int k)
        {
            a=k;
            b=0;
        }

    /*void setdata(int x, int y)
    {
        a=x;
        b=y;
    } */

    void showdata()
    {
        cout<<"na=" <<a<<" b="<<b;
    }
};
int main()
{
    complex c1;
    int x=5;
    c1=x;
    c1.showdata();
```



```
    return 0;
}
```

Output:

```
a=5 b=0
```

Conversion from Class Type to Basic Data Type

In this type of conversion, the programmer needs to explicitly tell the compiler how to perform conversion from class to basic type. Class type to primitive type can be implemented with casting operator. In this type, the left-hand operand is always of class type.

Example:

```
complex c1;
c1.setdata(3,4);
int x;
x=c1; //error
```

Syntax for casting operator

```
operator type()
{
    .....
return(type-data);
}
```

Program 14: Program to implement conversion from class type to primitive type.

```
#include<iostream>
using namespace std;
class complex
{
    private:
        int a,b;
    public:

void setdata(int x, int y)
{
    a=x;
    b=y;
}

void showdata()
{
    cout<<"\na=" <<a<<" b="<<b;
}

operator int()
{
    return(a);
}
};
int main()
{
```

```

    complex c1;
    c1.setdata(3,4);
    c1.showdata();
    int x;
    x=c1;
    cout<<"\nx="<<x;
    return 0;
}

```

Output:

```

a=3  b=4
x=3

```

Conversion from one class type o another class type

When an object of one class is assigned to object of another class, it is necessary to give clear cut instructions to the compiler about how to make conversion between these two user defined data types. The method must be instructed to the compiler. There are two ways to convert object data type from one class to another. Conversion from one class type to another class type can be implemented with

- Constructor
- Casting operator

Example:

```

class item
{
int a, b;
.....
};

class product
{
int m,n;
.....
};

int main()
{
Item i1;
product p1;
p1.setdata(3,4);
i1=p;
.....
}

```

Program 15: Program to implement conversion from one class type to another class type using constructor..

```

#include<iostream>
using namespace std;
class product
{

```

```

private:
    int m,n;
public:

void setdata(int x, int y)
{
    m=x;
    n=y;
}

int getM()
{
    return(m);
}
int getN()
{
    return(n);
}
};

class item
{
private:
    int a,b;
public:

item()
{
}

item(product p)
{
    a=p.getM();
    b=p.getN();
}

void showdata()
{
    cout<<"na=" <<a<<" b="<<b;
}
};

int main()
{
    item i1;
    product p1;
    p1.setdata(3,4);
    i1=p1;
    i1.showdata();
    return 0;
}

```

}

Output:**a=3 b=4****INHERITANCE: INTRODUCTION**

The mechanism of deriving a new class from an old class is called inheritance. The old class is referred to as base class and the new one is called the derived class. The derived class inherits some or all the properties from the base class.

1. It's a method of implementing reusability of Classes.
2. The Members of one Class can be accumulated in another Class through Inheritance.
3. The Class which acts as the source of providing its Members for Inheritance is called the Parent or Base Class. The Class that derives from or acquires the Members of Base Class is called the Child or Derived Class.
4. The Private members of any Class can Not be Inherited.
5. It adds some enhancement to the base class.

DEFINING DERIVED CLASSES

A class that uses inheritance to gain the behavior and data of another ('base') class is called derived class. A derived class is defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is as follows.

```
class derived class_name : visibility-mode base-class-name
{
    members of derived class;
    -----
};
```

The colon indicates that the derived-class-name is derived from the base-class-name. The visibility-mode is optional and, if present, may be either private or public. The default visibility-mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

Examples:

```
class ABC: private XYZ          //private derivation
{
members of  ABC
};
```

```
class ABC: public XYZ          //public derivation
{
members of  ABC
};
```

```
class ABC: protected XYZ      //protected derivation
{
members of  ABC
};
```

```
class ABC: XYZ           //private derivation by default
{
members of  ABC
};
```

Visibility modes

Public mode: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Note : The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example.

Example:

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
```

```
// y is private
// z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

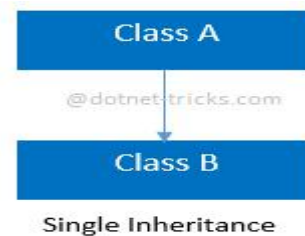
FORMS OF INHERITANCE

- i. Single Inheritance
- ii. Multiple Inheritance
- iii. Multilevel Inheritance
- iv. Multipath inheritance
- v. Hierarchical Inheritance
- vi. Hybrid Inheritance

Single Inheritance: -

In this inheritance, a derived class is created from a single base class.

In the given example, Class A is the parent class and Class B is the child class since Class B inherits the features and behavior of the parent class A.



Syntax:

```
class subclass_name : access_mode base_class
{
    //body of subclass
};
```

Program 16: Program to explain Single inheritance.

```
#include <iostream>
using namespace std;
class Vehicle {
public:
    Vehicle()
```

```

    {
        cout << "This is a Vehicle" << endl;
    }
};
class Car: public Vehicle{
};

```

```

// main function
int main()
{
    Car obj;
    return 0;
}

```

Output:

```
This is a Vehicle
```

Program 17: Program to explain Single inheritance.

```

#include<iostream>
using namespace std;
class A
{
    char name[20];
    int roll;
public:
    void getbase()
    { cout<<"enter roll and name:";
      cin>>roll>>name;
    }
    void showbase()
    { cout<<"\nname="<<name<<"\nroll="<<roll;
    }
};
class B:public A
{float h,w;
public:
    void getd()
    { getbase();
      cout<<"enter height and weight:";
      cin>>h>>w;
    }
    void showd()
    { showbase();
      cout<<"\nheight="<<h<<"\nweight="<<w;
    }
};
int main()
{
    B b;
    b.getd();
}

```

```

    b.showd();
return 0;
}

```

Output:

```

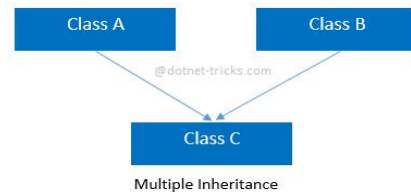
enter roll and name:1 rita
enter height and weight:5 55

name=rita
roll=1
height=5
weight=55

```

Multiple Inheritance: -

In this inheritance, a derived class is created from more than one base class. In the given example, class c inherits the properties and behavior of class B and class A at same level. So, here A and Class B both are the parent classes for Class C.

**Syntax:**

```

class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
    //body of subclass
};

```

Here, the number of base classes will be separated by a comma (',') and access mode for every base class must be specified.

Program 18: Program to explain multiple inheritance.

```

#include <iostream>
using namespace std;
class Vehicle {
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
class FourWheeler {
public:
    FourWheeler() {
        cout << "This is a 4 wheeler Vehicle" << endl;    };
};
class Car: public Vehicle, public FourWheeler {
};
int main() {
    Car obj;
    return 0; }

```

Output:

```

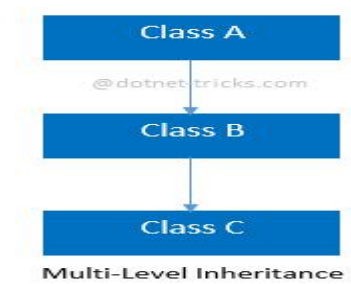
This is a Vehicle
This is a 4 wheeler Vehicle

```


Multilevel inheritance: -

In this inheritance, a derived class is created from another derived class.

In the given example, class c inherits the properties and behavior of class B and class B inherits the properties and behavior of class A. So, here A is the parent class of B and class B is the parent class of C. So, here class C implicitly inherits the properties and behavior of class A along with Class B i.e there is a multilevel of inheritance.

**Syntax:**

```

class A {.....};
class B: public A {.....};
class C: public B {.....};

```

Program 19: Program to explain multilevel inheritance.

```

#include <iostream>
using namespace std;
class Vehicle
{
    public:
        Vehicle()
        {
            cout << "This is a Vehicle" << endl;
        }
};
class fourWheeler: public Vehicle
{
    public:
        fourWheeler()
        {
            cout<<"Objects with 4 wheels are vehicles"<<endl;
        }
};
class Car: public fourWheeler{
    public:
        Car()
        {
            cout<<"Car has 4 Wheels"<<endl;
        }
};
int main()
{
    Car obj;
    return 0;
}

```

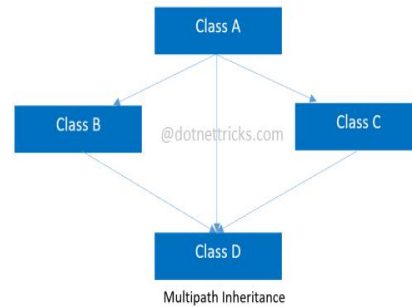
Output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

Multipath Inheritance:-

In this inheritance, a derived class is created from another derived classes and the same base class of another derived classes.

In the given example, class D inherits the properties and behavior of class C and class B as well as Class A. Both class C and class B inherits the Class A. So, Class A is the parent for Class B and Class C as well as Class D. So it's making it Multipath inheritance.



Program 20: Program to explain multipath inheritance.

```
#include <iostream>
using namespace std;
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
class Car: public Vehicle
{
};
class Bus: public Vehicle
{
};
class Truck: public Car, public Bus {
};
int main()
{
    Truck obj1;
    return 0;
}
```

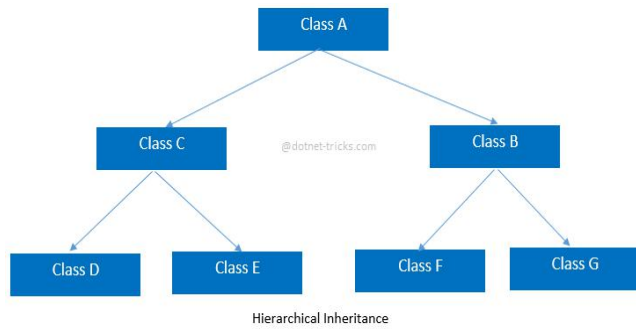
Output:

```
This is a Vehicle
This is a Vehicle
```

Hierarchical Inheritance: -

In this inheritance, more than one derived classes are created from a single base class and further child classes act as parent classes for more than one child classes.

In the given example, class A has two children: class B and class C. Further, class B and class C both are having two children - class D and E; class F and G respectively.



Program 21: Program to implement Hierarchical Inheritance.

```

#include <iostream>
using namespace std;

class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class Car: public Vehicle
{
};

class Bus: public Vehicle
{
};

int main()
{
    Car obj1;
    Bus obj2;
    return 0;
}
  
```

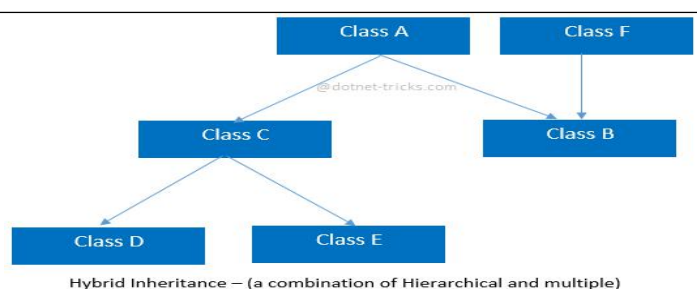
Output:

```

This is a Vehicle
This is a Vehicle
  
```

Hybrid inheritance:

This is combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance or Hierarchical and Multipath inheritance or Hierarchical, Multilevel and Multiple inheritance.

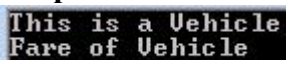


Program 22: Program for Hybrid Inheritance.

```

#include <iostream>
using namespace std;
class Vehicle
{
    public:
        Vehicle()
        {
            cout << "This is a Vehicle" << endl;
        }
};
class Fare
{
    public:
        Fare()
        {
            cout<<"Fare of Vehicle\n";
        }
};
class Car: public Vehicle
{
};
class Bus: public Vehicle, public Fare
{
};
int main()
{
    Bus obj2;
    return 0;
}

```

Output:


```

This is a Vehicle
Fare of Vehicle

```

AMBIGUITY IN SINGLE INHERITANCE

Ambiguity may arise in single inheritance. Consider the below example. In this case, the function in the derived class overrides the inherited function and, therefore, a simple call to display() by B type object will invoke function defined in B only. However, we may invoke the function defined in A by using the scope resolution operator to specify the class.

Program 23:

```

#include<iostream>
using namespace std;
class A
{public:

```

```

void display()
{
cout<<"A"<<endl;
}
};
class B: public A
{public:
void display()
{
cout<<"B"<<endl;
}
};
int main()
{
B b;
b.display();
b.A::display();
b.B::display();
return 0;
}

```

AMBIGUITY IN MULTIPLE INHERITANCE

We may face a problem in using the multiple inheritance, when a function with a same name appears in more than one base class. Consider the following two classes.

```

class M
{
public:
void display(void)
{
cout<<"class M\n";
}
};

class N
{
public:
void display(void)
{
cout<<"class N\n";
}
};

```

Which display() function is used by the derived class when we inherit these two classes? We can resolve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

```

class P: public M, public N
{
public:
void display(void) //overrides display() of M and N

```

```
{
M ::display();
}
};
```

We can now use the derived class as follows:

```
int main()
{
P obj;
obj.display();
}
```

VIRTUAL BASE CLASS

An object of multipath inheritance having two sets of grandparent class members. They are one from parent1 and the other from parent2. The duplication of inherited members due to multiple paths can be avoided by making the common base class as virtual base class while declaring the direct base classes as shown below:.

```
class A                      //grandparent
{
    -----
    -----
};
class B1 :  virtual public A    //parent1
{
    -----
    -----
};

class B2 :  virtual public A    //parent2
{
    -----
    -----
};

class C : public B1, public B2
    {
        // it will contain only one
        -----
        ----- // only one copy of A will be inherited
    };
```

Program 24: Program for virtual base class.

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

class Vehicle
{
public:
    Vehicle()
```

```

{
    cout << "This is a Vehicle" << endl;
}
};
class Car: virtual public Vehicle
{

};

class Bus: virtual public Vehicle
{

};

class Truck: public Car, public Bus {
};
int main()
{
    Truck obj1;
    return 0;
}

```

Output:

```
This is a Vehicle
```

OVERRIDING MEMBER FUNCTIONS

When both base class and derived class have a member function with same name and arguments (number and type of arguments). If you create an object of the derived class and call the member function which exists in both classes (base and derived), the member function of the derived class is invoked and the function of the base class is ignored. This feature in C++ is known as function overriding.

```

class Base
{
    ... ..
public:
    void getData();
    {
        ... ..
    }
};

class Derived: public Base
{
    ... ..
public:
    void getData();
    {
        ... ..
    }
};

int main()
{
    Derived obj;
    obj.getData();
}

```

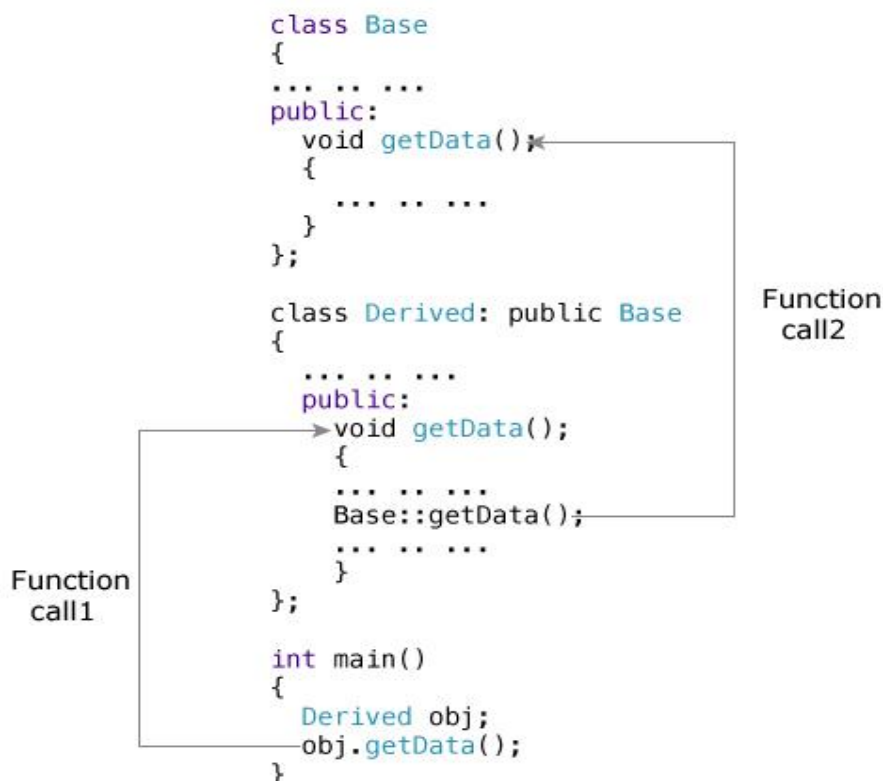
This function will not be called

Function call

How to access the overridden function in the base class from the derived class?

To access the overridden function of the base class from the derived class, scope resolution operator `::` is used. For example, If you want to access `getData()` function of the base class, you can use the following statement in the derived class.

```
Base::getData();
```



Difference between Overloading and Overriding

No	Overloading	Overriding
1	Relationship between methods available in the same class.	Relationship between a super class method and a subclass method.
2	Does not block inheritance from the super class.	Blocks inheritance from the super class.
3	Separate methods share (overload) the same name.	Subclass method replaces (overrides) the super class method.
4	Different method signatures.	Same method signatures.
5	May have different return types.	Must have matching return types.
6	May have different declared exceptions.	Must have compatible declared exceptions.

OBJECT SLICING

Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

Program 25: program for object slicing.

```
#include<iostream>
using namespace std;
class A
{
int a;
int b;
public:
A()
{
a=0;
b=0;
}
A(int x, int y)
{
a=x;
b=y;
}
void show()
{
cout<<"a:"<<a;
cout<<endl<<"b:"<<b;
}
};
class B: public A
{
int c;
public:
B(int x, int y, int z): A(x,y)
{
c=z;
}
void display()
{
show();
cout<<endl<<"c:"<<c;
}
};
int main()
{
B ob(12,23,48);
A ob1;
ob1=ob;
ob1.show();
}
```

Output:

```
a:12
b:23
```

ORDER OF EXECUTION OF CONSTRUCTORS AND DESTRUCTORS

When we use the inheritance, the constructor of the base class is called first and then the constructor of the derived class. Destructor are called opposite to the call of constructor. The destructor of derived class is called first and then the destructor of the base class.

Program 26: program to understand order of execution of constructors and destructors.

```
#include<iostream>
using namespace std;
class Person
{
public:
    Person()
    {
        cout<<"constructor of the person class called"<<endl;
    }
    ~Person()
    {
        cout<<"destructor of the person class called"<<endl;
    }
};

class Student: public Person
{
public:
    Student()
    {
        cout<<"constructor of the student class called"<<endl;
    }
    ~Student()
    {
        cout<<"destructor of the student class called"<<endl;
    }
};

int main()
{
    Student anil;
    return 0;
}
```

Output:

```
constructor of the person class called
constructor of the student class called
destructor of the student class called
destructor of the person class called
```

OBJECT COMPOSITION

Object composition refers to combining two or more different classes with purpose of creating new, more complex class. In case of composition, an object "owns" another object, rather than just use it, which means if main object will be destroyed, all internal objects should be destroyed as well.

The use of object in a class as data member is referred as “object composition”. Object composition is an alternative to class inheritance. Here new functionality is obtained by arranging or composing objects to support more powerful functionality. In this an object can be a collection of many other objects and the relationship is called a has-a relationship or containership. The has-a relationship occurs when an object of one class is contained in another class as a data member.

Example:

```
class B
{
.....
.....
};
class D
{
.....
B objb
};
```

Program 27: program to understand object composition..

```
#include<iostream>
using namespace std;
class B
{
public:
    int num;
    B()
    {
        num=0;
    }
    B(int a)
    {
        cout<<"constructor B(int a) is invoked"<<endl;
        num=a;
    }
};
class D
{
    int data1;
    B objb;
public:
    D(int a): objb(a)
    {
        data1=a;
    }
    void output()
    {
        cout<<"data in object of class D="<<data1<<endl;
        cout<<"data in member object of class B in class D="<<objb.num;
    }
};
int main()
```

```
{
D objd(10);
objd.output();
return 0;
}
```

Output:

```
constructor B(int a) is invoked
data in object of class D=10
data in member object of class B in class D=10
```

Composition Vs. Inheritance

<u>Sl. no</u>	<u>Composition</u>	<u>Inheritance</u>
<u>1</u>	Object composition refers to combining two or more different classes with purpose of creating new, more complex class.	Inheritance is used to create a new class (derived class) from an existing class(base class).
<u>2</u>	Bit harder to understand	Easier to understand
<u>3</u>	Loose coupling	Tightly coupled
<u>4</u>	Great flexibility	Less flexible
<u>5</u>	The constructor of the class D(derived) is invoked first and then the object of B is created.	The constructor of base class are first invoked before the constructor of the derived class.

DELEGATION

It is a way of making object composition as powerful as inheritance for reuse. In delegation two objects are involved in handling a request: a receiving object delegate operations to its delegate. Delegation shows that inheritance can be replaced with object composition as a mechanism for code reuse.

Example::

```
class book: public publication, public sales
```

```
{
//body of the book class
}
```

The above functionality can be achieved by composing objects of the classes publication and sales into the class book as follows:

```
class book
{
.....
publication pub;
sals market;
.....
}
```

Program 28: program to understand delegation..

```
#include<iostream>
```

```

using namespace std;
class publication
{
char title[40];
float price;
public:
void getdata()
{
cout<<"\t Enter title: ";
cin>>title;
cout<<"\t Enter price: ";
cin>>price;
}
void display()
{
cout<<"\t Title: "<<title<<endl;
cout<<"\t Price: "<<price<<endl;
}
};
class sales
{
private:
float publishsales[3];
public:
void getdata();
void display();
};
void sales::getdata()
{
int i;
for(i=0;i<3;i++)
{
cout<<"\t Enter sales of "<<i+1<<" month: ";
cin>>publishsales[i];
}}
void sales ::display()
{
int i;
int totalsales=0;
for(i=0;i<3;i++)
{
cout<<"\t Sales of "<<i+1<<" month: "<<publishsales[i]<<endl;
totalsales=totalsales+publishsales[i];
}
cout<<"\t Total sales: "<<totalsales<<endl;
}

class book
{
int pages;
public:
publication pub;
sales market;

```

```

void getdata()
{
    pub.getdata();
    cout<<"\t Enter no. of pages: ";
    cin>>pages;
    market.getdata();
}

void display()
{
    pub.display();
    cout<<"\t Number of pages: "<<pages<<endl;
    market.display();
}
};

int main()
{
    book book1;
    cout<<"Enter Book Publication Data"<<endl;
    book1.getdata();
    cout<<"Book Publication Data"<<endl;
    book1.display();
}

```

Output:

```

Enter Book Publication Data
    Enter title: u
    Enter price: 7
    Enter no. of pages: 5
    Enter sales of 1 month: 7
    Enter sales of 2 month: 7
    Enter sales of 3 month: 7
Book Publication Data
    Title: u
    Price: 7
    Number of pages: 5
    Sales of 1 month: 7
    Sales of 2 month: 7
    Sales of 3 month: 7
    Total sales: 21

```

Question Bank

Apr-May-2018

4. (a) What are public, private and protected inheritances? [2]
- (b) Explain operator overloading with example. [7]
- (c) Discuss ambiguity associated with multiple and multipath inheritance. [7]
- (d) Compare composition and inheritance. [7]

Nov-Dec-2017

4. (a) What is operator overloading ? [2]
(b) Explain inheritance and its type with example in detail. [7]
(c) What are the ambiguities in inheritance and how to resolve it? [7]
(d) Explain type conversion from basic type to class type and class type to basic type with one example. [7]

Apr-May-2017

4. (a) What is operator overloading ? [4]
(b) List the operators that cannot be overloaded. What may be the reasons for the same ? [7]
(c) What is virtual base class ? Explain with code. [7]
(d) Explain the following : [3½+3½=7]
(i) Multiple inheritance
(ii) Hybrid inheritance

Nov-Dec-2016

4. (a) What do you mean by overriding? 2
(b) Write a program to overload '+' operator to add two complex numbers. 7
(c) What are the various types of inheritance? Explain Hybrid inheritance with the help of example. 7
(d) What is data type conversion? Explain class type to basic type conversion with example. 7

Apr-May-2016

4. (a) Define polymorphism and inheritance. 2
- (b) Explain different types of inheritance in C++. 7
- (c) Write a C++ program to add two complex numbers using operator overloading concepts. 7
- (d) Explain the ambiguity associated with multiple inheritance. 7

Nov-Dec-2015

4. (a) Define type conversion in short. 2
- (b) How can we overload unary operator? 7
- (c) What is virtual base class? 7
- (d) Explain ambiguity in multipath and multiple inheritance. 7

Apr-May-2015

4. (a) What is the need of Operator Overloading? 2
- (b) Write a program to check whether given year is leap year or not using operator overloading. 7
- (c) What is Inheritance? Explain the type and need of inheritance and write the advantage and disadvantage of inheritance. 7
- (d) Explain the ambiguity between Multiple and Multipath inheritance. 7

Nov-Dec-2014

- Q. 4. (a) What do you mean by operator overloading in C++? 2
- (b) Explain the concept of inheritance in OOP with suitable example. 7

- (c) Discuss the ambiguity associated with Multiple and multipath inheritance. 7
- (d) Compare Composition and Inheritance. 7

Apr-May-2014

- Q. 4. (a) What is a scope resolution operator. 2
- (b) Explain operator overloading with example. 7
- (c) Explain inheritance with its different forms. 7
- (d) Write short notes on any two : 7
- (i) Virtual base class
 - (ii) Multipath inheritance
 - (iii) Object sliding
 - (iv) Type conversion