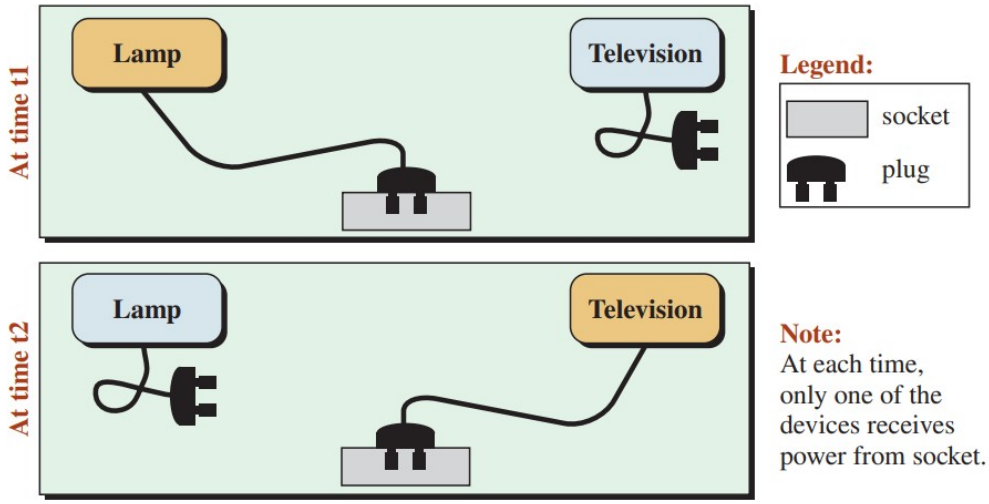
The term "**Polymorphism**" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

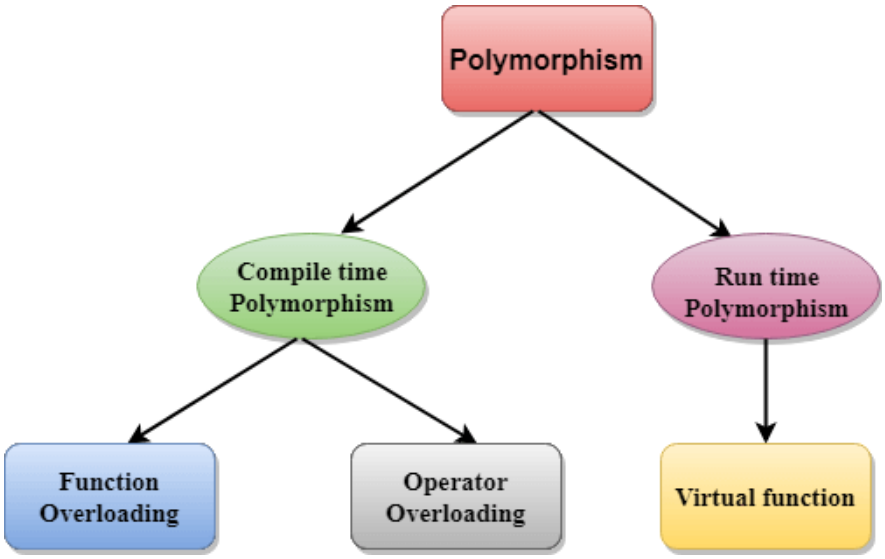
Polymorphism gives us the ability to write several versions of a function, each in a separate class. Then, when we call the function, the version appropriate for the object being referenced is executed. We see the same concept in our everyday language. We use one verb (function) to mean different things. For example, we say “open,” meaning to open a door, a jar, or a book; which one is determined by the context. Similarly, in C++, we can call a function named printArea to print the area of a triangle or the area of a rectangle.

To better understand the concept of polymorphism, we must first understand the concept of plug-compatible objects.



Assume, we have two electrical devices (such as a table lamp and a television set). We have only one socket that can supply power to one of these devices at a time. Each device has a plug that can be inserted in the only socket. At time t1, we plug the lamp into the socket. At time t2, we unplug the lamp and plug the TV set in the socket. We can do so because the two devices are plug-compatible; their plugs follow the same standard. The interesting point about plug-compatible devices is that all get the same thing (electrical power) from the socket, but each does a different task.

There are two types of polymorphism in C++:



Compile time polymorphism: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
class A                          //base class declaration.
  {
      int a;
      public:
      void display()
      {
           cout<< "Class A ";
       }
  };
class B : public A                    //derived class declaration.
{
    int b;
    public:
   void display()
   {
        cout<<"Class B";
```

```
  }
};
```

In the above case, the prototype of display() function is the same in both the base and derived class. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as run time polymorphism.

Run time polymorphism: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by function overriding, virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

```
#include <iostream>
using namespace std;
class Animal
{
	public:
		void eat()
		{
			cout<<"Eating...";
		}
};
class Dog: public Animal
{
	public:
		 void eat()
		{
			cout<<"Eating bread...";
		}
};
int main()
{
	Dog d = Dog();
	d.eat();
	return 0;
}
```

Run time polymorphism in C++ with two derived classes.

```
#include <iostream>
using namespace std;
class Shape {                                  //  base class
   public:
virtual void draw(){                          // virtual function
```

```
cout<<"drawing..."<<endl;
    }
};
class Rectangle: public Shape                  //  inheriting Shape class.
{
 public:
 void draw()
   {
      cout<<"drawing rectangle..."<<endl;
    }
};
class Circle: public Shape                     //  inheriting Shape class.

{
 public:
 void draw()
   {
     cout<<"drawing circle..."<<endl;
   }
};
int main(void) {
    Shape *s;                          //  base class pointer.
    Shape sh;                          // base class object.
       Rectangle rec;
        Circle cir;
      s=&sh;
     s->draw();
       s=&rec;
     s->draw();
    s=?
    s->draw();
}
```

Runtime Polymorphism with Data Members

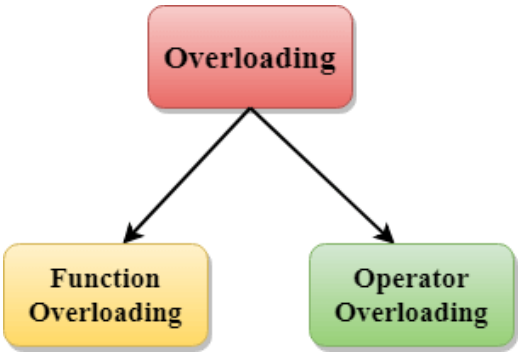
```
#include <iostream>
using namespace std;
class Animal {                                  //  base class declaration.
    public:
    string color = "Black";
};
class Dog: public Animal                        // inheriting Animal class.
{
 public:
    string color = "Grey";
};
int main(void) {
     Animal d= Dog();
    cout<<d.color;
}
```

OVERLOADING

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload methods, constructors, and indexed properties.

Types of overloading in C++ are:

- Function overloading
- Operator overloading



Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments.

It is only through these differences compiler can differentiate between the functions. The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

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

class Cal {
    public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C;                                  //     class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
   return 0;
}
```

Output:

```
r1 is : 42
r2 is : 0.6
```

Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as function overloading. When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

1. Type Conversion.
2. Function with default arguments.
3. Function with pass by reference.

1. Type Conversion

```
#include<iostream>
using namespace std;
void fun(int);
void fun(double);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(double j)
{
   std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
   fun(12);
   fun(1.2);
   return 0;
}
```

2. Function with default arguments

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
   fun(12);
   return 0;
}
```

3. Function with pass by reference

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
    int a=10;
     fun(a); // error, which f()?
return 0;
}
```

```
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

OPERATOR OVERLOADING

Operator overloading refers to overloading of one operator for many different purpose. Use of operator overloading permits us to see no difference between built-in data types and user defined data types. It is one of the powerful and fascinating features of the C++ which give additional meaning to built-in standard operators like +, -, *, /, >, <, < =, > = etc.

RULES OF OPERATOR OVERLOADING

1. Existing operators can only be overloaded, but the new operators cannot be overloaded.

2. The overloaded operator contains atleast one operand of the user-defined data type. All of the operands cannot be of basic types. If this is the case than function must be friend function of some class.

3. We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.

4. When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.

5. When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

6. You can change the meaning of the operator i.e., a + operator can be overloaded to perform multiplication operation or > operator can be overloaded to perform addition operation. But you cannot change the priority of the operators.

We can overload all the C++ operators except the following,

1. Class member access operators (., . *)
2. Scope resolution operator (: :)
3. Size operator (sizeof)
4. Conditional operator (?:)

The general form is:

```
return type class_name :: operator op (arrgument_list)
{
        //body of the function
}
```

Where,

return type is the type of value returned by the function and

class_name is the name of the class

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Operator functions must be either member functions (nonstatic) or friend functions.

The difference is a friend function will have only one argument for unary operators and two for binary operators. While a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not case with

friend functions, because arguments may be passed either by value or by reference. Operator functions are declared in the class using prototypes as follows:

```
vector operator+ (vector);                      // vector addition
vector operator- ( ) ;                          // unary minus
friend vector operator+ (vector, vector) ;      // vector addition
friend vector operator- (vector);               // unary minus
vector operator- (vector & a);                  // subtraction
int operator == (vector);                       // comparision
friend int operator== (vector,vector)           //comparision
```

The process of overloading involves :

a) Create a class that defines the data type that is to be used in the overloading operation.

b) Declare the operator function operator op () in the public part of the class. It may be either a member or friend function.

c) Define the operator function to implement the required operation.

Overloaded operator functions can be invoked by expressions such as.

op x or x op For unary operators

x op y //for binary operators.

op x or x op would be interpreted as

operator op (x) for friend functions

and x op y would be interpreted as

x.operator op (y) in member function and

operator op (x,y) in friend function.

OVERLOADING UNARY OPERATORS

i) In overloading unary operator, a friend function will have only one argument, while a member function will have no arguments.

ii) Let us consider the unary minus operator. A minus operator, when used as a unary takes just one operand.

iii) This operator changes the sign of an operand when applied to a basic data item.

Following example shows 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 variables.

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}

void operator ++()        {
         num = num+2;
      }
      void Print() {
         cout<<"Count : "<<num;
      }
};
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output

Count : 10

Overloading a unary minus operator using a friend function :

```
Friend void operator- (unary & u); //declaration
Void operator- (unary & u)            //definition
{
        u.x = -u.x;
        u.y = -u.y;
        u.z = -u.z;
}
```

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 changes made inside the operator function will not reflect in the called object. we can overload unary −, pre and post ++, pre and post – and unary +.

In case of overloading binary operators using member function of class left operand is responsible for calling the operator function and right operand was send as argument. In case of unary operator only one operand is there and this operand itself calls the overloaded operator function. Nothing is send as argument to the function. Its general syntax is (defined with the class)

```
return_type operator op ( )
{
// function code;
}

demo operator - ( )
{
        demo temp;
        temp.num = -num;
        return temp;
}

d1=-d; // equivalent to d1 = d.operator – ( );
```

OPERATOR OVERLOADING WITH BINARY OPERATOR

In overloading binary operator, a friend function will have two arguments, while a member function will have one argument.

```
#include <iostream>
using namespace std;
class A
{
   int x;
     public:
     A(){}
   A(int i)
   {
      x=i;
   }
   void operator+(A);
   void display();
};

void A :: operator+(A a)
{
   int m = x+a.x;
   cout<<"Result: "<<m;
}
int main()
{
   A a1(5);
   A a2(4);
   a1+a2;
   return 0;
}
```

Output

Result : 9

Thus in overloading binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument.

When overloading binary operator using operator function as class member function, the left side operand must be an object of the class and right side operand may be either a built-in type or user defined type.

```
#include <iostream>
using namespace std;
class demo_sum
{
private :
    int num;
    static int count;
public :
void input( )
{
cout<<"Enter the number for object"<<++count<<endl;
cin>>num;
}
int operator + (demo_sum temp)
{
return(num + temp.num);
}
void show( )
{
cout<<"The num is "<<num<<endl;
}
};
int demo_sum::count;
int main()
{
demo_sum d1, d2;
d1.input( );
```

```
d2.input( );
d1.show( );
d2.show( );
int sum=d1+d2;
cout<<"The sum of two object’s num is "<<sum<<endl;
return 0;
}
```

Above program using another method

```
#include <iostream>
using namespace std;
class demo_sum
{
private :
    int num;
    static int count;
public :
void input( )
{
cout<<"Enter the number for object"<<++count<<endl;
cin>>num;
}
int operator + (demo_sum temp)
{
return(num + temp.num);
}
void show( )
{
cout<<"The num is "<<num<<endl;
}
};
int demo_sum::count;
int main()
{
```

```
demo_sum d1, d2;
d1.input( );
d2.input( );
d1.show( );
d2.show( );
int sum=d1+d2;
cout<<"The sum of two object's num is "<<sum<<endl;
return 0;
}
```

OVERLOADING USING FRIEND FUNCTION

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

For example, to overload a binary + we declare a function as a class member function as :

```
demo operator + (demo A)
```

And in the main we write as d1+d2. As the operator function is a member function of class left side operand must be an object of the class. This is must as internally d1+d2 is treated as d.operator + (d2).

To write the same function using friend:

```
friend demo operator + (demo A, demo B);
```

Friend function is not a member function of the class so it cannot be called using an object of the class. So in case of binary + overloaded using friend d1+d2 is interpreted as operator + (d1, d2). That is no object or no operand calls the function and both the operand are send as argument.

```
//overloading unary ++ operator overloaded using class
    demo operator ++( );
//The same operator ++ overloaded using friend can be written as
    friend demo operator ++ (demo d);
```

And d2 = d1++ is equivalent to d2 = operator ++ (d1).

In case of expression written as d1 +d2, d1+20 d1*3, class operator functions will work but in case of expression like 20 +d1, 10*d1 etc., class operator functions will not work as left operand must be an object of the class. In such situations we can use friend function.

There are certain operators which cannot be overloaded. They are:

S.No.	*Operators*	*Name of Operator*
1.	: :	Scope resolution operator
2.	.	Dot membership operator
3.	.*	Pointer to member operator
4.	? :	Conditional operator
5.	sizeof	The size of operator

There are operators which cannot be overloaded using friend function. They are given as :

S.No.	*Operators*	*Name of Operator*
1.	=	Assignment operator
2.	->	Pointer to member operator
3.	[ ]	Subscript operator
4.	( )	Function call operator.

```
#include <iostream>
#include <conio.h>

using namespace std;

class Complex
{
private:
   int x, y;

public:
   Complex(int x = 0, int y = 0) : x(x), y(y) {}
   Complex operator+(const Complex &rhs)
   {
      Complex c3;
      c3.x = rhs.x + x;
      c3.y = rhs.y + y;
      return c3;
   }
```

```
    Complex operator-(const Complex &rhs)
    {
        Complex c4;
        c4.x = x - rhs.x;
        c4.y = y - rhs.y;
        return c4;
    }
    void display()
    {
        cout << "The number is " << x << " + " << y << "i" << endl;
    }
};

int main()
{
    Complex c1(2, 3), c2(4, 6);
    c1.display();
    c2.display();
    Complex c3 = c1 + c2;
    c3.display();

    Complex c4 = c1 - c2;
    c4.display();
    return 0;
}
```

Assignment Operator Overloading

```
#include <iostream>
#include <conio.h>

using namespace std;

class Test
{
private:
    int *x;

public:
    Test(int val = 0) : x(new int(val)) {}
    ~Test();
    void setX(int val) { *x = val; }
    void print()
    {
        cout << "output: " << *x << endl;
    }
    Test &operator=(const Test &rhs)
    {
        if (this != &rhs)
            *x = *rhs.x;
```

```
        return *this;
    }
};

Test::~Test()
{
}

int main()
{
    Test t1(10);
    Test t2;
    t2 = t1;
    t1.setX(20);
    t1.print();
    t2.print();

    return 0;
}
```

Overloading [] operator in c++ (Array Subscript Operator)

Points:

1. Array subscript Operator can not be used to chk out of bound classes.
2. Array Subscript operator can not be friend fuction and others too(-> ,() , =)

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

class Point
{
    int arr[2]; // x -> 0 , y -> 1

public:

    Point(int x = 0, int y = 0)
    {
        arr[0] = x;
        arr[1] = y;
    }
    void Print()
    {
        cout << "x: " << arr[0] << " y: " << arr[1] << endl;
    }

    int &operator[](int pos)
    {
        if (pos == 0)
```

```
        {
            return arr[0];
        }
        else if (pos == 1)
        {
            return arr[1];
        }
        else
        {
            cout << "Out of bound case. " << endl;
            exit(0);
        }

        {
            /* code */
        }
    }
};



int main()
{
    Point p1(2, 3);
    p1.Print();
    p1[0] = 11;
    p1[1] = 12;
    p1.Print();
    p1[0] = 20;
    p1.Print();
    return 0;
}
```

Type Conversions

Converting data types of a variable into other data type is known as type conversion. We have studied it earlier when we convert int to float, char to int, double to int etc. Compiler also does implicit type conversion. But all we studied involved all built-in types. In this section, we study type conversion with respect to user data type viz. class.

Consider the following statement that adds two objects and then assign the result to a third object.

```
v3=v1+v2                    // v1,v2 and v3 are class type objects
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. In the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand.

Since the user defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types.

We divide the study of type conversion into three parts.

1. Conversion from Built-in types to class type.
2. Conversion from class type to built-in types.
3. Conversion from one class type to another.

1. Conversion from Built-in Types to Class Type

When we want to convert basic built-in types into class types, the simple methods is to define parameterized constructor which takes an argument of basic type which you want to convert to class type.

For example, when you want an int to be converted to class type say demo, define a one argument constructor type int in demo class as :

```
demo (int)
{
        constructor body;
}
demo d=10
```

Type Conversion INT to Class Type

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

class demo
{
   int data;
public:
   demo(int x)
   {
      data = x;
      show();
   }
   void show()
   {
      cout << "data=" << data << endl;
   }
};
int main()
```

```
{
    int num;
    cout << "Enter the num" << endl;
    cin >> num;
    demo d = num;
    return 0;
}
```

Output

```
12
data=12
```

String type to class type conversion

```
#include <iostream>
#include <ctype.h>
#include <string.h>
class demo
{
        char str[20];
  public :
        demo(char x[ ])
        {
        strcpy(str,x);
        }
        int countV( )
        {
                int i=0,count=0;
                char ch;
        while(str[i]!=0)
        {
                ch=toupper(str[i]);
        switch(ch)
        {
                case ‘A’ :
```

```
                case 'E' :
                case 'I' :
                case 'O' :
                case 'U' : count++;
        }
        i++;
}
return count;
}
};

void main( )
{
        demo d="NMIMS University";
        int c =d.countV( );
        cout<<"Number of vowels = "<<c<<endl;
}
```

Output

Number of Vowels =5

Type conversion Float to Class Type

```
#include <iostream.h>
#include <iomanip.h>

class demo
{
        float num;
  public :
        demo(float x)
        {
        num = x;
        }
        int int_part( )
```

```
        {
                return int (num);  //45
        }
        float real_part( )
        {
                return (num-int_part( ));   //0.56
        }
};
int main( )
{
        demo d= 45.56;
        cout<<“Integer Part = “<<d.int_part( )<<endl;
        cout<<“RealPart=”<<setprecision(2) <<d.real_part( )<<endl;
}
```

OUTPUT :

Integer Part = 45

Real Part=0.56

Type Conversion Long Int to Class Type

```
#include <iostream.h>
class CountDigit
{
        unsigned long int num; //4294967295
  public :
        CountDigit(unsigned long int x)
        {
                num = x;   //1234567890
        }
void count( )
{
        int temp [10]= {0};
        int r,i;
while(num!=0)
{
```

```
        r=num%10;
        temp[r]++;
        num = num/10;
}
cout<<“DIGIT \tFREQ\n”;
for(i=0;i<10;i++)
        cout<< i<<“\t”<<temp[i]<<endl;
}
};
void main( )
{
        CountDigit cd=1234567890;
        cd.count( );
}
```

OUTPUT :

```
DIGIT FREQ
0       1
1       1
2       1
3       1
4       1
5       1
6       1
7       1
8       1
9       1
```

2. Conversion from Class Type to Built-in Types

Here we study reverse of what we have studied in the first part. To convert class type to built-in type the method to be adopted is that simply defined an operator function by the name of data type you want class data type to be converted to. The general syntax is :

```
operator data_type ( )
{

}
```

For example, assume conversion function is defined in the class demo :

```
operator int ( )
{
        int x;
        computing steps;
        return x;
}
```

Type Conversion, Class to Int

```
#include <iostream.h>
class demo
{
        int num;
   public :
        demo(int x)
        {
                num=x;
        }
operator int( )
{
        return num*num;
```

```
}
};
int main( )
{
        int k;
        cout<<“Enter the number for k :=”;
        cin>>k;
        demo d(k);
        int s=d;
        cout<<“Square =”<<s<<endl;
}
```

OUTPUT :

Enter the number for k : =15

Square =225

Type Conversion, Class to Float

```
#include <iostream.h>
class circle
{
        float rad;
  public :
        circle(float x)
        {
                rad = x;
        }
operator double( )
{
        return 3.14 * rad * rad;
}
};
int main( )
{
        circle d(4.5);
```

```
        double area=d;
        cout<<“Area of circle = “<<area<<endl;
}
```

OUTPUT :

Area of circle = 63.585

Type Conversion, Class to Char

```
#include <iostream.h>
class Number
{
        int num;
        public :
Number(int x)
{
        num = x;
}
operator char ( )
{
        if(num<0)
        return ‘N’;
        else if(num ==0)
        return ‘Z’;
        else return ‘P’;
}
};
int main( )
{
        int n;
        cout<<“Enter a number”<<endl;
        cin>>n;
```

```
        Number d(n);
        char ch =d;
        switch(ch)
        {
                case ‘P’ :
                cout<<“POSITIVE”<<endl;
        break;
        case ‘N’ :
        cout<<“NEGATIVE”<<endl;
        break;
        case ‘Z’ :
        cout<<“ZERO”<<endl;
        break;
}
}
```

OUTPUT :

(First run)

Enter a number

23

POSITIVE

(Second run)

Enter a number

-23

NEGATIVE

(Third run)

Enter a number

0

ZERO

Type Conversion, Class to Char

```
#include <iostream.h>
#include <string.h>
class Rev
{
        char str[25];
  public :
        Rev( )
        {
        cout<<“ENTER A STRING “<<endl;
        cin.getline(str,25);
        }
        operator char*( );
};
Rev : :operator char*( )
{
        static char s[25];
        strcpy(s,strrev(str));
        return s;
}
int main( )
{
        Rev R;
        char *t = R;
        cout<<“Reverse String is “<<t<<endl;
}
```

OUTPUT :

ENTER A STRING

Thinkpalm

Reverse String is mlapknihT

3. Conversion from one Class Type to Another

In programming situations when we want to convert object of one class to other class type we can follow either of the two approach we have seen in part one and part two. Suppose we have a class first second and we want to convert an object of class first type into second type. In the main we will be writing as :

```
first f; second s;
s=f;
```

Note here we want an object f of class to be converted to class second type. In the first method we can write an operator function in the second in the source class (assume source is first and destination is second as we convert from first to second) as :

```
Operator second ( )
{

    function body;
}
```

This operator function must return an object of class second type. In the second method for the same conversion s=f we can write a one argument constructor in the destination class second which takes an object of first class as argument as :

```
second (first fobj)
{
    Constructor body;
}
```

When compiler see **s = f** it automatically look for any conversion routine which can convert an object of first type into second type. As the above constructor was written in the class second it will be called.

Type Conversion One Classs to Another Class Type

Converting an object of class first into class type second

```
#include <iostream.h>
class second
{
        int sx;
  public :
        second( ){}
        second(int x)
        {
                sx=x;
        }
        int & getsx( )
        {
                return sx;
        }
        void shows( )
        {
                cout<<“sx=”<<sx<<endl;
        }
        };
class first
{
        int fx;
  public :
        first( ){}
        first(int x)
        {
        fx = x;
```

```
        }
        operator second( )
        {
                second temp;
                temp.getsx( )=fx*fx;
                return temp;
        }
        void showf( )
        {
                cout<<“fx=”<<fx<<endl;
        }
};

int main( )
{
        first f(20);
        second s(30);
        cout<<“Before conversion”<<endl;
        f.showf( );
        s.shows( );
        s=f;
        cout<<“After conversion”<<endl;
        f.showf( );
        s.shows( );
}
```

OUTPUT :

Before conversion

fx=20

sx=30

After conversion

fx=20

sx=400

Second method of converting object of one class type to another

```
#include <iostream.h>
class first
{
        int fx;
  public :
        first( ){}
        first (int x)
        {
                fx=x;
        }
        int getfx( )
        {
                return fx;
        }
        void showf( )
        {
                cout<<“fx=”<<fx<<endl;
        }
};

class second
{
        int sx;
  public :
        second( ){}
        second(int x)
        {
                sx=x;
        }
        second(first obj)
        {
                sx=obj.getfx( )*obj.getfx( );
        }
```

```
        void shows( )
        {
cout<<“sx=”<<sx<<endl;
}
};

int main( )
{
        first f(100);
        second s(200);
        cout<<“Before Conversion”<<endl;
        f.showf( );
        s.shows( );
        s=f;
        cout<<“After Conversion”<<endl;
        f.showf( );
        s.shows( );
}
```

OUTPUT :

Before Conversion

fx=100

sx=200

After Conversion

fx=100

sx=10000

Third method

```
// operator = function
#include <iostream.h>
class first
{
        int fx;
  public :
        first( ){}
        first(int x)
        {
                fx=x;
        }
        int getfx( )
        {
                return fx;
}
void showf( )
{
cout<<“fx=”<<fx<<endl;
}
};

class second
{
        int sx;
        public :
        second( ){}
second(int x)
{
        sx=x;
}
void operator=(first obj)
{
```

```
        sx=obj.getfx( )*obj.getfx( );
}
void shows( )
{
        cout<<“sx=”<<sx<<endl;
}
};
int  main( )
{
        first f(100);
        second s(200);
        cout<<“Before Conversion”<<endl;
        f.showf( );
        s.shows( );
//s=f           s.operator=(f)
        cout<<“After Conversion”<<endl;
        f.showf( );
        s.shows( );
}
```

OUTPUT :

Before Conversion
fx=100
sx=200

After Conversion
fx=100
sx=200

Converting Hour to Minute and Second

```
#include <iostream.h>
class Hour
{
        unsigned int hr;
  public :
        Hour(unsigned int x)
        {
                hr=x;
        }
int gethr( )
{
        return hr;
}
void showh( )
{
cout<<“Hour=”<<hr<<endl;
}
};

class secmin
{
        unsigned int sec, min;
  public :
        secmin( ){}
        secmin(Hour obj)
        {
        min=obj.gethr( )*60;
        sec=min*60;
        }
        void showsm( )
        {
```

```
        cout<<“Mins=”<<min<<endl;
        cout<<“Secs=”<<sec<<endl;
        }
};
int main( )
{
        Hour H(5);
        secmin sm;
        sm=H;
        H.showh( );
        sm.showsm( );
}
```

OUTPUT :

Hour=5

Mins=300

Secs=18000