AMBITION GURU

Object-Oriented programming in C++ (CSIT 202)

II semester, BScCSIT

Compiled by Ankit Bhattarai
Ambition Guru College

Syllabus

AMBITION GURU

Unit	Contents	Hours	Remarks
1.	Introduction to C++ and OOP	3	
2.	Classes and objects	7	
3.	**Operator Overloading & Type Conversion**	**7**	
4.	Inheritance	7	
5.	Polymorphism and Virtual Functions	7	
6.	Templates and Exception Handling	7	
7.	I/O Stream	8	

Practical Works

Credit hours : 3

AMBITION GURU

Unit 3
Operator Overloading and Type Conversion
(7 hrs.)

Introduction to operator overloading, overloading unary and binary operators, Friend functions, friend class, Type conversion: Basic-to-class, Class-to-basic, Class-to-class.

Operator Overloading :

- The process of redefining or extending the meaning of the existing operators while using with user defined data type is called *operator overloading*.
- The operator overloading feature of C++ is one of the methods of realizing polymorphism.

Following are the operators that cannot be overloaded in C++:

1. Member access operators (.)
2. Scope resolution operator(::)
3. Conditional operator(?:)
4. Size operator (sizeof( ))
5. Run-time type information operator (typeid)
6. Pointer to member access (*)

Operator Overloading:

Consider the following statements

```
int a,b;
int  c = a + b;
complexx c1,c2;
complexx c = c1 + c2;
```

- Here **+ operator** is said to be overloaded as it is used with basic type data a and b as well as user defined type c1 and c2.
- Other examples can be c1+5,c1++,++c1 etc. where c1 is the object of type complexx.

Rules for operator overloading:

i. Only the existing operators can be overloaded, new operators cannot be created.

ii. The overloaded operator must have at least one operand that is of user defined type.

iii. We cannot change the basic meaning of the operators that is to say we cannot redefine (+) operator to subtract one value from the another.

iv. Unary operator overloading through member operator function takes no argument where as overloading through non-member operator function(friend function) takes one argument.

v. Binary operator overloading through member operator function takes one argument where as overloading through non-member operator function takes two arguments.

vi. We cannot use friend function to overload the following operators:

1. = (assignment operator)
2. ( ) (function call operator)
3. [ ] (subscripting operator or index operator)
4. -> (arrow operator)

General syntax of operator overloading

```
class class_name{

   // private data members;

   public:

   return_type operator operator_symbol(no argument or argument);

   Or, // member operator function

   friend return_type operator operator_symbol(argument(s)); // friend function

};
```

```
return_type class_name::operator operator_symbol( no argument or argument)
{
    //Body of member operator function.
}
```

Or,

```
return_type operator operator_symbol(argument(s))
{
   //Body of non-member operator function(friend function).
}
```

Here,

return_type is the return type of the function.

operator is a keyword.

operator_symbol is the operator we want to overload. Like: +, <, -, ++, etc.

arguments is the arguments passed to the function.

AMBITION GURU

Example 1:
Add Two Complex Number

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

class Complex {
private:
    float real;
    float imag;

public:
    Complex(float r = 0, float i = 0) {
        real = r;
        imag = i;
    }

 // Operator overloading of +
    Complex operator + (Complex c) {
        Complex temp;
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
        return temp;
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};
```

Example 1:
Add Two Complex Number

```
int main() {
    Complex c1(2.5, 3.5), c2(1.5, 2.5), c3;

    c3 = c1 + c2;  // Using overloaded +

    cout << "Sum: ";
    c3.display();

    return 0;
}
```

The operator Keyword

How do we teach a normal C++ operator to act on a user-defined operand?

- The keyword operator is used to overload the ++ operator in this declarator:

```
void operator ++ ()
```

- The return type (void in this case) comes first, followed by the keyword operator, followed by the operator itself (++), and finally the argument list enclosed in parentheses (which are empty here).
- This declarator syntax tells the compiler to call this member function whenever the ++ operator is encountered, provided the operand (the variable operated on by the ++).

Operator Arguments

- In main() the ++ operator is applied to a specific object, as in the expression ++c1.

Yet operator++() takes no arguments. What does this operator increment?

- It increments the data in the object of which it is a member.
- Since member functions can always access the particular object for which they've been invoked, this operator requires no arguments.

Friend function in C++

Friend function in C++

- In C++, a ***friend function*** is a function that is **not a member of a class**, but still has access to the class's **private and protected** members.
- It is useful when you want a function (global or member of another class) to access internal members of a class without making them public.
- A friend function is declared inside the class using the keyword friend.
- It is not a member function of the class.
- It can access private and protected members of the class.
- Can be a normal function, or a member of another class.

Friend function in C++

Why use it?

- To allow an external function to **access private data** of a class.
- Useful when you want a function to **operate on multiple objects** (possibly from different classes).

Friend function in C++

AMBITION GURU

Syntax of Friend Function

Inside class:

```
friend returnType functionName(arguments);
```

Outside class:

```
returnType functionName(arguments) {
    // can access private members
}
```

AMBITION GURU

Friend function in C++

Example 1: Add Two Numbers Using Friend Function

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

class Number {
private:
    int value;

public:
    Number(int v = 0) {
        value = v;
    }

    // Declare friend function
    friend int add(Number a, Number b);
};

// Define friend function
int add(Number a, Number b) {
    return a.value + b.value;
}

int main() {
    Number n1(10), n2(20);

    cout << "Sum = " << add(n1, n2) << endl;

    return 0;
}
```

Friend function in C++

AMBITION GURU

Example 2: Friend Function Accessing Two Different Classes

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

class B;  // Forward declaration

class A {
private:
    int a;

public:
    A(int val = 0) {
        a = val;
    }

    // Declare friend function
    friend int add(A, B);
};
```

Friend function in C++

AMBITION GURU

Example 2: Friend Function Accessing Two Different Classes

```
class B {
private:
    int b;

public:
    B(int val = 0) {
        b = val;
    }

    // Declare friend function
    friend int add(A, B);
};

// Define the friend function
int add(A objA, B objB) {
    return objA.a + objB.b;
}
```

Friend function in C++

AMBITION GURU

Example 2: Friend Function Accessing Two Different Classes

```
int main() {
    A obj1(10);
    B obj2(20);

    cout << "Sum = " << add(obj1, obj2) << endl;

    return 0;
}
```

Output:
Sum = 30

Unary operator overloading:

- If the existing operator is used within the single operand it is called unary operator.
- Unary prefix and postfix operators can be overloaded with the help of member operator function as well as friend function.

Syntax for overloading unary prefix and unary postfix operators through member operator function:

```
class class_name{
  //private data members;
  public:
  return_type operator operator_symbol ( );  //for prefix
  return_type operator operator_symbol(int);  //for postfix
};
```

```
return_type class_name::operator operator_symbol() //prefix
{
   //Body of member operator function.
}

return_type class_name::operator operator_symbol(int) //postfix
{
   //Body of member operator function.
}
```

Syntax for unary prefix and postfix overloading with the help of friend function:

```
class class_name
{
//private data_members;
public:
friend return_type operator operator_symbol(class_name); //prefix
friend return_type operator operator_symbol(class_name,int); //postfix
};
```

```
return_type operator operator_symbol (class_name object_name)//prefix
{
//Body of friend function.
}


return_type operator operator_symbol(class_name object_name,int)//postfix
{
//Body of friend function
}
```

```
// Overload ++ when used as prefix
#include <iostream>
using namespace std;
class Count {
   private:
   int value;
   public:
    // Constructor to initialize count to 5
    Count(){
        value=5;
    }
    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
```

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

int main()
{
    Count count1;
    // Call the "void operator ++ ()" function
    ++count1;
    count1.display();
    return 0;
}
```

Output:

```
Count: 6
```

```
// Overload ++ when used as prefix and postfix
#include <iostream>
using namespace std;
class Count {
   private:
    int value;
   public:
    // Constructor to initialize count to 5
    Count(){
        value = 5;
        }
    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
```

```
// Overload ++ when used as postfix
    void operator ++ (int) {
        value++;
    }
 void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;
    // Call the "void operator ++ (int)" function
    count1++;
    count1.display();
    // Call the "void operator ++ ()" function
         ++count1;
        count1.display();
        return 0;
}
```

Output:

Count: 6
Count: 7

```
#include <iostream>
using namespace std;
class Count {
   int value;
   public:
   Count() // Constructor to initialize count to 5
   {
     value = 5;
   }

   Count operator ++ () // Overload ++ when used as prefix
   {
       Count temp;
        // Here, value is the value attribute of the calling object
        temp.value = ++value;
        return temp;
   }
```

```
    // Overload ++ when used as postfix
     Count operator ++ (int)
     {
         Count temp;
         // Here, value is the value attribute of the calling object
         temp.value = value++;
         return temp;
     }
     void display() {
         cout << "Count: " << value << endl;
     }
};
```

```
int main()
{
    Count count1, result;
    // Call the "Count operator ++ ()" function
    result = ++count1;
    result.display();
    // Call the "Count operator ++ (int)" function
    result = count1++;
    result.display();
    return 0;
}
```

Output:

Count: 6
Count: 6

AMBITION GURU

Example program showing overloading of unary prefix operator(++) using member operator function and unary prefix(--) using friend function.

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

counter(int c)
{
  count=c;
}

int ret_value( )
{
   return count;
}
void operator ++( )
{
  ++count;
}
friend counter operator --(counter &);
};
```

```
counter operator --(counter &c)
{
counter temp;
temp.count= -- c.count;
return temp;
}
int main( )
{

counter c1(10);

counter c2;
++c1;//c1.operator ++( );
cout<<"The value of c1 is:"<<c1.ret_value( )<<endl;
c2=--c1;//c2=operator--(c1);
cout<<"The value of c1 is:"<<c1.ret_value( )<<endl;
cout<<"The value of c2 object is:"<<c2.ret_value( )<<endl;
 return 0;
}
```

Output:

```
The value of c1 is:11
The value of c1 is:10
The value of c2 object is:10
```

Nameless temporary objects

- When we want to return an object from member function of class without creating an object, for this: we just call the constructor of class and return it to calling function and there is an object to hold the reference returned by constructor.
- This concept is known as **nameless temporary objects**, using this we are going to implement a C++ program for **pre-increment operator overloading**.

```
#include <iostream>
using namespace std;
class Sample
{
     int count;
     public:
     Sample() //default constructor
     {
        count = 0;
     }

     Sample(int c) //parameterized constructor
     {
        count = c;
     }
```

```
Sample operator++() //Operator overloading function definition
     {
          ++count;
          //returning count of Sample
          //There is no new object here,
          //Sample(count): is a constructor by passing value of count
          //and returning the value (incremented value)
          return Sample(count);
     }

     //printing the value
     void printValue()
     {
          cout<<"Value of count : "<<count<<endl;
     }
};
```

```
int main()
{
    int i = 0;
    Sample S1(100), S2;
    for(i=0; i< 4; i++)
    {
        S2 = ++S1;
        cout<<"S1 :"<<endl;
        S1.printValue();
        cout<<"S2 :"<<endl;
        S2.printValue();
    }
    return 0;
}
```

Output:

```
S1 :
Value of count : 101
S2 :
Value of count : 101
S1 :
Value of count : 102
S2 :
Value of count : 102
S1 :
Value of count : 103
S2 :
Value of count : 103
S1 :
Value of count : 104
S2 :
Value of count : 104
```

Overloading Binary Operator

- The operator which is used between two operands is called binary operator.
- The binary operator can be overloaded with the help of member operator function as well as non-member operator function(friend function).

Syntax of overloading of binary operator with the help of member operator function:

```
class class_name
{
 //private
data_members;
public:
return_type operator operator_symbol(argument1)
{
  //Body of the member operator function.
}
};
```

// Overloading of the binary operator +.This program adds two complex numbers

```
#include <iostream>
using namespace std;
class Complexx {
    float real,imag;
   public:
   Complexx() // Constructor to initialize real and imag to 0
    {
       real = 0; imag =0;
     }
    void input() {
        cout << "Enter real and imaginary parts respectively: ";
        cin >> real;
        cin >> imag;
    }
```

// Overload the + operator

```
    Complexx operator + (Complexx obj) {
        Complexx temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }

   void output()
   {
        if (imag < 0)
            cout << "Output Complex number: " << real << imag << "i";
        else
            cout << "Output Complex number: " << real << "+" << imag << "i";
    }
};
```

```
int main() {
    Complexx complex1, complex2, result;
    cout << "Enter first complex number:\n";
    complex1.input();
    cout << "Enter second complex number:\n";
    complex2.input();
   // complex1 calls the operator function
   // complex2 is passed as an argument to the function
    result = complex1 + complex2;
    result.output();
    return 0;
}
```

Output:

Enter first complex number:

Enter real and imaginary parts respectively: 1

2

Enter second complex number:

Enter real and imaginary parts respectively: 6

7

Output Complex number: 7+9i

```
#include<iostream>
using namespace std;
class Money
{
int rupees, paisa;

public:
void input( )
{
 cout<<"Enter the values for rupees and paisa:"<<endl;
 cin>>rupees>>paisa;
}
void display( )
{
 cout<<rupees<<"Rupees and"<<paisa<<"Paisa"<<endl;
}
```

```
Money operator +(Money m){
Money temp;
   int s=(paisa+rupees*100)+(m.paisa+m.rupees*100);
   temp.rupees=s/100;
   temp.paisa=s%100;
   return temp;
}
Money operator -(Money m){
   Money temp;
   int d=(paisa+rupees*100)-(m.paisa+m.rupees*100);
   if(d< 0)
       d= -d;
   temp.rupees=d/100;
   temp.paisa=d%100;
   return temp;
}};
```

```
int main( )
{
   Money m1,m2,m3,m4;
   m1.input( );
   m2.input( );
   m1.display( );
   m2.display( );
   m3=m1+m2;//m3=m1.operator+(m2);
   m4=m1-m2;//m4=m1.operator-(m2);
   cout<<"After the addition of two money objects result=";
   m3.display( );
   cout<<"After the subtraction of two money objects result=";
   m4.display( );
return 0;
}
```

Output:

```
Enter the values for rupees and paisa:
1
50
Enter the values for rupees and paisa:
5
60
1Rupees and50Paisa
5Rupees and60Paisa
After the addition of two money objects result=7Rupees and10Paisa
After the subtraction of two money objects result=4Rupees and10Paisa
```

Note: While overloading binary operator through member operator function the left hand side of the operand is used to invoke the member operator function while right hand side of the operand is passed as an argument in the member operator function.

Syntax of binary operator overloading through non-member operator function or friend function:

```
class class_name
{
//private data_members;
public:
friend return_type operator operator_symbol(arg1,arg2);
};

return_type operator operator_symbol(arg1,arg2)
{
//Body of non-member operator function(friend function).
}
```

Example program of binary operator overloading through non-member operator function(friend function)

```
#include<iostream>
using namespace std;
class Distance
{
int feet,inches;
public:
void input( )
{
cout<<"Enter the values in feet and inches:"<<endl;
cin>>feet>>inches;
}
```

```
 void display ( )
{
 cout<<feet<<" Feet and "<<inches<<" Inches"<<endl;
}
 friend void operator +(Distance,Distance);
 friend void operator-(Distance,Distance);
};

 void operator +(Distance d1,Distance d2)
{
 int s=(d1.inches+d1.feet*12)+(d2.inches+d2.feet*12);
 int f=s/12;
 int i= s%12;
 cout<<f<<" Feet and"<<i<<" Inches"<<endl;
}
```

```
void operator -(Distance d1,Distance d2)
{
int d=(d1.inches+d1.feet*12)-(d2.inches+d2.feet*12);
if(d<0)
d= -d;
int f=d/12;
int i= d%12;
cout<<f<<" Feet  and"<<i<<" Inches"<<endl;
}
```

```
int main( )
{
Distance d1, d2;
d1.input( );
d2.input( );
d1.display( );
d2.display( );
cout<<"After the addition of two distance objects result="<<endl;
d1+d2; //operator+(d1,d2)
cout<<"After the subtraction of two distance objects result="<<endl;
d1-d2; // operator-(d1,d2)
return 0;
}
```

Output:

```
Enter the values in feet and inches:
1
11
Enter the values in feet and inches:
2
10
1 Feet   and 11 Inches
2 Feet   and 10 Inches
After the addition of two distance objects result=
4 Feet   and 9Inches
After the subtraction of two distance objects result=
0 Feet and 11Inches
```

Note:

- In case of binary operator overloading through non-member operator function(friend function) both the left and right hand side of the operand are passed as an argument in the non-member operator function or friend function.

Mandatory use of friend function while overloading binary operators:

Let us consider the statements, c2=c1+5;

c2=5+c1;

Where, c1 and c2 are objects of type complexx.

- The first statement c2=c1+5 can be overloaded by help of both member operator function as well as friend function however the statement c2=5+c1 can only be overloaded by the help of a friend function.
- This is because we know that in case of binary operator overloading by the help of member operator function the left hand side of the operand is used to invoke the member operator function which should be the object of a class.

- But in the statement c2=5+c1 the left hand side of the operand is built in data type so it cannot be used to invoke the member operator function.
- So in this case the friend function becomes mandatory as friend function is invoked as the normal function and both left hand side operand and right hand side operand are passed as an arguments.

Example

WAP to achive operations like c2=c1+5 and c2=5+c1 where c1 and c2 are objects of type complexx.

```
#include<iostream>
using namespace std;
class complexx
{
float real,imag;
public:
void input( ){
   cout<<“Enter the real and imaginary part:”<<endl;
   cin>>real>>imag;
}
void display( ){
   cout<<“(“<<real<<“ , “<<imag<<“)”<<endl;
}
```

```
complexx operator +( int p){
   complexx temp;
   temp.real=real+p;
   temp.imag=imag;
    return temp;
}
friend complexx operator+(int,complexx);
};

complexx operator+(int p,complexx c){
complexx temp;
temp.real=p+c.real;
temp.imag=c.imag;  return temp;
}
```

```
int main( )
{
complexx c1,c2;
c1.input( );
c1.display ( );
c2=c1+5; //c2=c1.operator+(5);
cout<<"Addition using member operator function the result=";
c2.display( );
c2=5+c1; //c2=operator+(5,c1);
cout<<"Using non-member operator function(friend function) the result=";
c2.display( );
return 0;
}
```

Output:

Enter the real and imaginary part:
3
4
(3 , 4)
Addition using member operator function the result=(8 , 4)
Addition using non-member operator function(friend function) the result=(8 , 4)

Write a program to compare magnitude of complex numbers by overloading <,>,== and != operators.

Write a program to compare magnitude of complex numbers by overloading <,>,== and != operators.

AMBITION GURU

```
#include<iostream>
#include<math.h>
using namespace std;
class complexx
{
float real,imag;
public:
void input()
{
cout<<"Enter the value for real and imaginary part:"<<endl;
cin>>real>>imag;
}
```

```
void display( )
{
   cout<<"("<<real<<" ,"<<imag<<")"<<endl;
}

int operator <(complexx c)
{
 float mag1=sqrt(real*real+imag*imag);
 float mag2=sqrt(c.real*c.real+c.imag*c.imag);
 if(mag1<mag2)
    return 1;
 else
    return 0;
}
```

```
int operator >(complexx c){
 float mag1=sqrt(real*real+imag*imag);
 float mag2=sqrt(c.real*c.real+c.imag*c.imag);
 if(mag1>mag2)
    return 1;
 else
    return 0;
}

int operator ==(complexx c){
 float mag1=sqrt(real*real+imag*imag);
 float mag2=sqrt(c.real*c.real+c.imag*c.imag);
 if(mag1==mag2)
    return 1;
 else
    return 0;
}
```

```
int operator !=(complexx c)
{
 float mag1=sqrt(real*real+imag*imag);
 float mag2=sqrt(c.real*c.real+c.imag*c.imag);
 if(mag1!=mag2)
    return 1;
 else
    return 0;
}
};

int main( )
{
 complexx c1,c2;
 c1.input( );
 c2.input( );
```

```
c1.display( );
c2.display( );
if(c1<c2) //if(c1.operator<(c2))
 cout<<"First object is smaller than second object"<<endl;

if(c1>c2) //if(c1.operator>(c2))
 cout<<"First object is greater than second object"<<endl;

if(c1==c2) //if(c1.operator==(c2))
 cout<<"Both objects are equal"<<endl;

if(c1!=c2) //if(c1.operator!=(c2))
 cout<<"Both objects are not equal"<<endl;
return 0;
}
```

Output:

```
Enter the value for real and imaginary part:
1
3
Enter the value for real and imaginary part:
2
4
(1 ,3)
(2 ,4)
First object is smaller than second object
Both objects are not equal
```

AMBITION GURU

Output:

```
Enter the value for real and imaginary part:
2
3
Enter the value for real and imaginary part:
2
3
(2 ,3)
(2 ,3)
Both objects are equal
```

C++ program to copy marks of Student 2 to Student 1

```
// C++ program to copy marks of Student 2 to Student 1
#include <iostream>
using namespace std;
class student {
    int english,math;
public:
    student(int e, int m)
    {
        english = e;
        math = m;
    }
    void operator=(student s)
    {
        english = s.english;
        math = s.math;
    }
```

```
  // method to display marks
    void marks()
    {
        cout << "English: " << english << ", Math: " << math<< endl;
   }
};

int main()
{
    student s1(6, 2), s2(5, 10);
    cout << "Student 1 marks : ";
    s1.marks();
    cout << "Student 2 marks : ";
    s2.marks();
   // use assignment operator
    s1 = s2;
    cout << endl;
```

```
    cout << "Student 1 marks : ";
     s1.marks();
     cout << "Student 2 marks : ";
     s2.marks();
     return 0;
}
```

Output:

Student 1 marks : English: 6, Math: 2
Student 2 marks : English: 5, Math: 10

Student 1 marks : English: 5, Math: 10
Student 2 marks : English: 5, Math: 10

Index([ ]) operator overloading:

The index([ ]) operator which is used with the array to access the array elements can be overloaded and it is considered as binary operator.

Syntax:

```
class class_name
{
//private
 data_members;
public:
return_type operator [ ](int_type argument)
{
//Body of the member operator function;
}
};
```

- As the array is accessed using the subscript(index) that is of int type so the paramter to the member operator function should be of int type.

Example program

```
#include<iostream>
#include<cstdlib>
using namespace std;
class Array{
int a[4];
public:
Array( int b[4]){
for(int i=0;i<4;i++)
a[i]=b[i];
}
```

```
void operator [ ]( int index)
{
 if(index<0 || index>=4)
 {
    cout<<"Out of range:"<<endl;
    exit(0);
 }
else
 {
    cout<<"The element is:"<<a[index]<<endl;
 }
}
};
```

```
int main( )
{
 int b1[ ]={1,2,3,4};
 Array A(b1);
 for(int i=0;i<4;i++)
 A[i]; //A.operator[
](i);
 return 0;
}
```

Disadvantage of using operator overloading in C++.

- The encapsulation feature is violated in object oriented if we use non member operator function(friend function) in operator overloading. This is the disadvantage in operator overloading since it violates the data hiding concept.

Data conversion (Type conversion)

Data conversion(Type conversion)

- The = operator will assign a value from one variable to another. Let us consider the statement like

```
var1=var2;
```

•Where var1 and var2 are variables. We may also have noticed that = assigns the value of one user defined object toanother, provided they are of the same type, in statements like

```
dist3=dist1+dist2;
```

- Where the result of the addition, which is type Distance is assigned to another object of type Distance,dist3.

- Normally, when the value of one object is assigned to another of the same type, the values of the all the data member items are copied in to the new object.
- The compiler does not need any special instructions to use = for the assignment of user defined objects such as Distance objects. (Implicit type conversion)
- What if the assignment operator is used for the different type i.e. int to user defined or one user defined to another user defined etc. For these conversions, compiler will not perform it but users need to tell it to perform(Explicit type conversion)

- In C++ under the following conditions an explicit(programmer intervention) conversion is needed .

1)Conversion from basic type to class type

2)Conversion from class type to basic type

3)Conversion from one class type to another class type.

Conversion from basic type to class type

Conversion from basic type to class type

AMBITION GURU

In C++, to convert basic type data to class type one argument parameterized constructor is used in a class where argument of basic type is passed as an argument in a constructor.

Example program:

```
#include<iostream>
using namespace std;
class time
{
        int hr, minute, sec;
        public:
        time( )
        {
        hr=0;
        minute=0;
        sec=0;
        }
```

```
time(int t) /*one argument parameterized  constructor for basic type to class type data conversion*/
{
    hr=t/3600;
    minute=(t%3600)/60;
    sec=(t%3600)%60;
}
void display( )
{
cout<<hr<<" Hour "<<min<<" Minute and "<<sec<<" Second";
}
};
```

```
int main( )
{
int duration=3695;
time t;
t=duration; /*coversion from basic type data to class type;
OR
time t(duration);
time t=duration; */
t.display( );
return 0;
}
```

Output:

1 Hour 1 Minute and 35 Second

Conversion from class type to basic type

Conversion from class type to basic type

AMBITION GURU

- For conversion from class type to basic type data casting operator function is used in a class.

Syntax for casting operator function:

```
class class_name
{
//private data members;
public:
operator data_type( )
{
//Body of the casting operator function;
}
};
```

Following are the characteristics of casting operator function:

1)It must be the member of a class.

2)It must not contain any argument.

3)It has a return statement.

Example program for conversion from class type to basic type data:

```
#include<iostream>
using namespace std;
class celcius
{
float temp;
public:
```

```
void get_temperature( )
{
     cout<<"Enter the value of temperature in celcius:"<<endl;   cin>>temp;
}

void display( )
{
     cout<<"The given value of temperature in celcius is:"<<temp<<endl;
}

operator float( )
{
     float fer=(temp*1.8)+32;
     return fer;
}
};
```

```
int main( ){
    celcius cel;
    cel.get_temperature( );
    cel.display( );
    float fer1;
    fer1=cel;
    //fer1=cel.operator float( );
    cout<<"The equivalent temperature in fahrenheit is:"<<fer1;
    return 0;
}
```

AMBITION GURU

Output:

Enter the value of temperature in celcius:
37.8
The given value of temperature in celcius is:37.8
The equivalent temperature in fahrenheit is:100.04

Conversion from one class type to another classtype

Conversion from one class type to another classtype

AMBITION GURU

- In C++ one class type to another class type conversion can be performed in two ways:

  i.By using one argument parameterized constructor.

  ii.By using casting operator function.

- If one argument parameterized constructor is used then it should lie in a destination class.
- If casting operator function is used then it should lie in source class.

An example program to show conversion of object of type polar to the object of type cartesian using one argument parameterized constructor.

AMBITION GURU

```
#include<iostream>
#include<math.h>
using namespace std;
class polar
{
float radius, angle;
public:
polar( )
{
}
polar( float r,float a)
{
radius=r; angle=a;
}
float ret_radius( )
{
return radius;
}
float ret_angle( )
{
return angle;
}
```

```
void display( )
{
    cout<<"("<<radius<<" , "<<angle<<")"<<endl;
}
};
class cartesian
{
    float xco,yco;
public:
    cartesian( )
    {
        xco=0.0;
        yco=0.0;
    }
```

```
cartesian(polar p)//one argument parameterized constructor for conversion
{
    xco=p.ret_radius( )*cos(p.ret_angle( ));
    yco=p.ret_radius( )*sin(p.ret_angle( ));
}
void display( )
{
    cout<<"("<<xco<<" ,"<<yco<<")"<<endl;
}
};
```

```
int main( )
{
    polar pol(5.5,45.5);
    cartesian cart;
    cart=pol;//cartesian cart(pol);
    cout<<"The given polar is:";
    pol.display( );
    cout<<"The equivalent cartesian is:";
    cart.display( );
    return 0;
}
```

Output:

The given polar is:(5.5 , 45.5)

The equivalent cartesian is:(0.291877 ,5.49225)

An example program to show conversion of object of type polar to object of type cartesian using casting operator function.

AMBITION GURU

```
#include<iostream>
#include<math.h>
using namespace std;
class cartesian;
//forward declaration
class polar
{
        float radius, angle;
        public:
        polar( )
        {
                radius=0.0; angle=0.0;
        }
```

```
polar(float r, float a)
{
    radius=r;
    angle=a;
}
void display( )
{
    cout<<"("<<radius<<" ,"<<angle<<")"<<endl;
}
    operator cartesian( );//casting operator function
};
```

```
class cartesian
{
    float xco,yco;
    public:
    cartesian( )
    {
        xco=0.0; yco=0.0;
    }
    cartesian( float x,float y)
    {
        xco=x; yco=y;
    }
```

```
void display ( )
{
    cout<<"("<<xco<<" ,"<<yco<<")"<<endl;
}
};
polar::operator cartesian( )
{
    float x=radius*cos(angle);
    float y=radius*sin(angle);
    cartesian temp(x,y);
    return temp;
}
```

```
int main( )
{
     polar pol(5.5,45.5);
     cartesian cart;
     cart=pol; //cart=pol.operator cartesian( );
     cout<<"The given polar is :";
     pol.display( );
     cout<<"The equivalent cartesian is:";
     cart .display( );
     return 0;
}
```

Explicit Constructor

AMBITION GURU

- We have seen that one argument parameterized constructor which takes parameter as basic type is used to convert from basic typedata to class type data.
- Such one argument parameterized constructor can also made as explicit by adding keyword explicit before the constructor name known as explicit constructor and such constructor should be called explicitly.
- An explicit constructor is a constructor that cannot be used for implicit type conversions. It prevents the compiler from automatically performing conversions when constructing objects or calling functions that take the constructed object as a parameter.

Example program:

```
#include<iostream>
using namespace std;
class time
{
int hr, min, sec;
public:
time( )
{
 hr=0;
 min=0;
 sec=0;
}
```

```
explicit time( int t)  //explicit constructor
{
  hr=t/3600;
  min=(t%3600)/60;
  sec=(t%3600)%60;
}
void display( )
 {
    cout<<hr<<"Hour"<<min<<"Minute and"<<sec<<"Second"<<endl;
 }
 };
```

```
int main( )
{
        int duration=3695;
        time t(duration);// ok explicit call to constructor.
        //time t=duration; error.
        //time t;
        //t=duration; error
        // OR
        // time t=time(duration);ok explicit call to constructor.
        t.display( );
        return 0;
}
```

THANK YOU

Any Queries ?