UNIT -2

CONSTRUCTOR'S AND OPERATOR OVERLOADING

1. Explain constructor overloading with example. Create a class Time and display hour, minute and seconds using constructor overloading. (16)

Constructors are the special type of member function that initializes the object automatically when it is created Compiler identifies that the given member function is a constructor by its name and return type. Constructor has same name as that of class and it does not have any return type.

```
class temporary
  {
     private:
       int x;
       float y;
     public:
       temporary(): x(5), y(5.5)       /* Constructor  */
          {
               /* Body of constructor */
          }
       .... ...  ....
 }
 int main()
   {
     Temporary t1;
     .... ... ....
   }
```

Working of Constructor

In the above pseudo code, temporary() is a constructor. When the object of class temporary is created, constructor is called and x is initialized to 5 and y is initialized to 5.5 automatically.

You can also initialize data member inside the constructor's function body as below. But, this method is not preferred.

```
temporary(){
  x=5;
  y=5.5;
}
```

/* This method is not preferred. /*

Use of Constructor in C++

Suppose you are working on 100's of objects and the default value of a data member is 0. Initializing all objects manually will be very tedious. Instead, you can define a constructor which initializes that data member to 0. Then all you have to do is define object and constructor will initialize object automatically. These types of situation arises frequently

while handling array of objects. Also, if you want to execute some codes immediately after object is created, you can place that code inside the body of constructor.

Constructor Example

```
/*Source Code to demonstrate the working of constructor in C++ Programming */
/* This program calculates the area of a rectangle and  displays it. */
#include <iostream>
using namespace std;
class Area
{
   private:
     int length;
     int breadth;

   public:
     Area(): length(5), breadth(2){ }   /* Constructor */
     void GetLength()
     {
        cout<<"Enter length and breadth respectively: ";
        cin>>length>>breadth;
     }
     int AreaCalculation() {  return (length*breadth);  }
     void DisplayArea(int temp)
     {
        cout<<"Area: "<<temp;
     }
};
int main()
{
   Area A1,A2;
   int temp;
   A1.GetLength();
   temp=A1.AreaCalculation();
   A1.DisplayArea(temp);
   cout<<endl<<"Default Area when value is not taken from user"<<endl;
   temp=A2.AreaCalculation();
   A2.DisplayArea(temp);
   return 0;
}
```

Explanation

In this program, a class of name Area is created to calculate the area of a rectangle. There are two data members length and breadth. A constructor is defined which initializes length to 5 and breadth to 2. And, we have three additional member functions GetLength(), AreaCalculation() and DisplayArea() to get length from user, calculate the area and display the area respectively.

When, objects A1 and A2 are created then, the length and breadth of both objects are initialized to 5 and 2 respectively because of the constructor. Then the member function GetLength() is invoked which takes the value of length and breadth from user for object A1. Then, the area for the object A1 is calculated and stored in variable temp by calling AreaCalculation() function. And finally, the area of object A1 is displayed. For object A2, no data is asked from the user. So, the value of length will be 5 and breadth will be 2. Then, the area for A2 is calculated and displayed which is 10.

Output

```
Enter length and breadth respectively: 6
7
Area: 42
Default Area when value is not taken from user
Area: 10
```

Constructor Overloading

Constructor can be overloaded in similar way as function overloading. Overloaded constructors have same name(name of the class) but different number of argument passed. Depending upon the number and type of argument passed, specific constructor is called. Since, constructor are called when object is created. Argument to the constructor also should be passed while creating object. Here is the modification of above program to demonstrate the working of overloaded constructors.

```
/* Source Code to demonstrate the working of overloaded constructors */
#include <iostream>
using namespace std;
class Area
{
   private:
     int length;
     int breadth;

   public:
     Area(): length(5), breadth(2){ }         // Constructor without no argument
     Area(int l, int b): length(l), breadth(b){ } // Constructor with two argument
     void GetLength()
     {
        cout<<"Enter length and breadth respectively: ";
        cin>>length>>breadth;
     }
     int AreaCalculation() {  return (length*breadth);  }
     void DisplayArea(int temp)
     {
        cout<<"Area: "<<temp<<endl;
     }
};
int main()
```

```
{
   Area A1,A2(2,1);
   int temp;
   cout<<"Default Area when no argument is passed."<<endl;
   temp=A1.AreaCalculation();
   A1.DisplayArea(temp);
   cout<<"Area when (2,1) is passed as arguement."<<endl;
   temp=A2.AreaCalculation();
   A2.DisplayArea(temp);
   return 0;
}
```

Explanation of Overloaded Constructors

For object A1, no argument is passed. Thus, the constructor with no argument is invoked which initializes length to 5 and breadth to 2. Hence, the area of object A1 will be 10. For object A2, 2 and 1 is passed as argument. Thus, the constructor with two argument is called which initializes length to l(2 in this case) and breadth to b(1 in this case.). Hence the area of object A2 will be 2.

Output

Default Area when no argument is passed.
Area: 10
Area when (2,1) is passed as arguement.
Area: 2

Default Copy Constructor

A object can be initialized with another object of same type. Let us suppose the above program. If you want to initialize a object A3 so that it contains same value as A2. Then, this can be performed as:

```
....
int main() {
  Area A1,A2(2,1);
  Area A3(A2);    /* Copies the content of A2 to A3 */
    OR,
  Area A3=A2;     /* Copies the content of A2 to A3 */
}
```

TIME class:

```
#include<iostream.h>
 #include<conio.h>
 class Time
 {
  public:
  int Hour,Minutes,Seconds;
```

```
 public:
Time()
{
int Hour=10;
int Minutes=10;
int Seconds=10;
cout<<Hour<<"\t"<<Minutes<<"\t"<<Seconds<<endl;
}
/* Time(float hr=10,double min=25,int sec=56)
{
float Hour=hr;
double Minutes=min;
int Seconds=sec;
cout<<Hour<<"\t"<<Minutes<<"\t"<<Seconds<<endl;
}*/
Time(float hr)
{
float Hour=hr;
int Minutes=0;
int Seconds=0;
cout<<Hour<<"\t"<<Minutes<<"\t"<<Seconds<<endl;
}

Time(float hr,int min, int sec)
{
int Hour=hr;
int Minutes=min;
int Seconds=sec;
cout<<Hour<<"\t"<<Minutes<<"\t"<<Seconds<<endl;
 }

// Time(int i,int j)
 //{
 //Hour=i;
 //Minutes=j;
 //Seconds=0;

// Time(Time &t1)
// {
//    int Hour=t1.hr;
//    int Minutes=t1.min;
//    int Seconds=t1.sec;
//  }
  void Display()
 {
```

```
   cout<<Hour<<"\t"<<Minutes<<"\t"<<Seconds<<endl;

  }
 };

void main()
{
clrscr();
//Time t;
float h1;
int min1;
int sec1;
cout<<"Enter Hour:"<<endl;
cin>>h1;
cout<<"Enter Minutes:"<<endl;
cin>>min1;
cout<<"Enter Seconds:"<<endl;
cin>>sec1;
//Time t;
Time t1(h1,min1);
// Time t(9,29,59);
//Time t2(t);
Time t4(12.0,2,45);
// Time t2(t1);
// t1.Display();
Time t5(12);
 getch();
}
```

Copy Constructor:

```
#include<iostream.h>
#include<conio.h>
class Copyconst
{
 public:
 int count;
 Copyconst(int t)
 {
   count=t;
   count++;
   cout<<count;
 }
 Copyconst(Copyconst &i)
```

```
{
 count=i.count;
}
void show()
{
cout<<"\t"<<count<<endl;
}
};
void main()
{
  clrscr();
  Copyconst c1=Copyconst(20);
  //c1(20);
  Copyconst c2(c1);
  //c2(c1);
  c2.show();
  //c1=50;
  c1.show();
  getch();
}
```

2. Explain the use of explicit constructors with an example program. (10)

A constructor that takes a single argument is, by default, an implicit conversion operator that converts its argument to an object of its class. In order to avoid such implicit conversions, a constructor that takes one argument can be declared explicit. To create an explicit constructor, we need to precede the constructor name by the keyword explicit, for example,

```
class Student
{
public:
explicit Student(char *temp);    //disallow implicit conversion
};
```

Let us consider the following program,

```
#include<iostream.h>
#include<conio.h>
class Student
{
 private:
    char *Name;
public:
   Student(char *temp)   //one argument constructor
   {
         Name=temp;
   }
};
void main()
{
   Student s1;
   s1=”Rohit”;                  //calling the one argument constructor
}
```

Here the class student has one argument constructor. When the statement:

s1=”Rohit”;

is invoked by the compiler, it calls the one argument constructor, which constructs a student object. Then the compiler performs an implicit conversion to convert from string data type to student object. To avoid such implicit conversions, a constructor that takes one argument can be declared explicit.

“explicit constructor is a constructor when it is invoked a temporary object will be created and it is initialized with the provided value then a copy of temporary object is copied to actual object.”

Example:

```
#include<iostream.h>

#include <conio.h>

Class Student

{

  private:

     char *Name;

 public:

    explicit Student(char *temp)

    {

          Name=temp;

    }

};

Void main()

{

  Student s1=Student(“Michael”);              //calling the explicit constructor

   //Student s2;

//   s2 = “Michael”;                          //conversion not possible

}
```

3. Write a program to overload a unary operator as a member function and also explain the operator overloading. (10)

   "The mechanism of redefining the existing operator to a user defined data type is called as operator overloading."

   Rules of operator overloading:

   - Only the existing operator can be overloaded.
   - The overloaded operator must have at least one operand that is of user defined data type.
   - The basic meaning of the operator should not be changed. That is, if the operator to be overloaded to is + operator then it should be used only for addition operation, not for other arithmetic operations or any other.
   - Overload operators follow the syntax and semantics rules of original operators. They cannot be overridden.
   - Except the function call operator (), no other operator can have a default argument.

All operators having a single argument are unary operators. When we overload these operators as member functions, we do not need to pass any argument explicitly. The "this" pointer pointing to invoking object is passed as an implicit argument.

Example: increment – unary operator overloading as member functions

```
#include<iostream.h>
#include<conio.h>
class Unaryplusplusop
{
 int a,b;
 public:
   Unaryplusplusop(int x,int y)
   {
   a=x;
   b=y;
   }
```

```
  void operator++()
  {
    cout<<"pre increment";
    ++a;
    ++b;
  }
  void operator ++(int t)
  {
    cout<<"post increment";
    a++;
    b++;
  }
  void show()
  {
    cout<<"\n"<<a;
    cout<<"\n"<<b;
  }
  };
void main()
{
  clrscr();
  Unaryplusplusop u1(10,20);
  ++u1;
  u1.show();
  Unaryplusplusop u2(-12,3);
  ++u2;
  u2.show();
```

```
  Unaryplusplusop u3(20,5);
  u3++;
  u3.show();
  getch();
 }
```

Output:

```
Pre increment
11        21
Pre increment
-11       4
Post increment
21        5
```

Example-2: unary minus – member function

```
#include<iostream.h>
#include<conio.h>
class Unaryminusminusop
{
 int a,b;
 public:
  Unaryminusminusop(int x,int y)
  {
  a=x;
  b=y;
  }
  void operator--()
  {
```

```
    --a;
    --b;
  }
  void show()
  {
    cout<<"\n"<<a;
    cout<<"\n"<<b;
  }
  };
void main()
{
  clrscr();
  Unaryminusminusop u1(10,20);
  --u1;
  u1.show();
  Unaryminusminusop u2(-12,3);
  --u2;
  u2.show();
  getch();
}
```

Output:

```
9       19
-13     2
```

4. Write a program for overloading unary ++ operator as friend function. (8)

```
#include<iostream.h>
#include<conio.h>
class UnaryFriend
{

 int a,b;
 public:
  UnaryFriend()
 {}
  UnaryFriend(int x,int y)
  {
  a=x;
  b=y;
  }
  friend UnaryFriend operator++(UnaryFriend);
  friend UnaryFriend operator++(UnaryFriend,int);
  void show()
  {
  cout<<a<<endl;
  cout<<b<<endl;
  }
 };
 UnaryFriend operator++(UnaryFriend u1)
 {
  UnaryFriend temp;
  temp.a=++u1.a;
```

```
   temp.b=u1.b+1;
   return temp;
 }
 UnaryFriend operator++(UnaryFriend u,int t)
{
  UnaryFriend uf;
   u.a++;
   uf.b=u.b+t;
   return u;
 }
void main()
{
  clrscr();
  UnaryFriend u2(10,20);
  UnaryFriend u1;
  u1=++(u2);
  u1.show();
  UnaryFriend u3(20,25);
u1=(u3)++;
u1.show();
  getch();
}
```

Output:

11

21

21

25

5. Write a program to perform complex number addition, subtraction, multiplication and division.

```
#include <iostream.h>
#include<conio.h>
class Complex
{
 private:
 float real,imag;
 public:
 Complex()
 {}
 Complex(float r, float i)
 {
  real=r;
  imag=i;
 }
 Complex operator+(Complex);
 Complex operator-(Complex);
 Complex operator *(Complex);
 Complex operator /(Complex);
 void showdata()
 {
  cout<<real<<"+i"<<imag;
 }
 void getdata();
 };
 Complex Complex::operator +(Complex b2)
 {
  Complex temp;
  temp.real=real+b2.real;
  temp.imag=imag+b2.imag;
  return temp;
 }
Complex Complex::operator -(Complex b2)
 {
  Complex temp;
  temp.real=real-b2.real;
  temp.imag=imag-b2.imag;
  return temp;
 }
Complex Complex::operator *(Complex b2)
 {
  Complex temp;
  temp.real=real*b2.real+imag*b2.imag;
```

```
  temp.imag=real*b2.imag+imag*b2.real;
  return temp;
 }
Complex Complex::operator /(Complex b2)
 {
  Complex temp;
  float qt;
  qt=b2.real*b2.real+b2.img*b2.img;
  temp.real=(real*b2.real+imag*b2.imag)/qt;
  temp.imag=(imag*b2.real-real*b2.imag)/qt;
  return temp;
 }

 void Complex::getdata()
 {
  cout<<"enter real value";
  cin>>real;
  cout<<"enter imaginary value";
  cin>>imag;
 }
void main()
{
 clrscr();
 Complex b1,b2,b3;
 b1.getdata();
 b2.getdata();
 //Complex b1(20.2,3.1),b2(10.5,2.8),b3;
 b3=b1+b2;
 b3.showdata();
 b3=b1-b2;
 b3.showdata();
 b3=b1*b2;
 b3.showdata();
 b3=b1/b2;
 b3.showdata();
 getch();
}
```

Output:

```
Enter real value
10.2
Enter imaginary value
2.1
Enter real value
2.2
```

Enter imaginary value
3.4

12.4+i5.5
8+i-1.3
29.58+i39.299999
1.803658+i-1.832927

6. Explain different types of type converters with an example. (16)

Type conversion takes place between the incompatible data types. There are three types of data conversions. They are:

- Conversion from built-in data type to an object
- Conversion from object to a built-in data type
- Conversion from one object to another object type

Any type conversion involving a class is defined either

- By a conversion constructor
- By a conversion operator function

The conversion performs type conversion by converting any given type to the type of the current class.

A conversion operator function performs conversion in the opposite direction, that is, it converts an object of the current class to another type. The general form of this function is,

```
operator type_name()
{
  Function_body;
}
```

A conversion function is a function that satisfies the following conditions.

- It must be a member function
- It must not specify a return type
- It must not have any argument

Conversion from built-in data type to object:

Conversion from basic data type to object type can be done with the help of constructors. A constructor takes a single argument whose type is to be converted.

```
#include<iostream.h>

#include<conio.h>

class Sample

{

 int x;

public:

Sample()

{}

Sample(int a)

{

 x=a;

}

void Printdata()

{

 cout<<x;

}

};

void main()

{

 Sample s1;

 s1=5;

s1.Printdata();

}
```

Here in the above program, the statement

s1=5;

converts built-in data type into the object s1 by invoking the conversion constructor.

Conversion from object to a built-in data type:

Using the conversion function, conversion from object of a class to a built-in type can be done.

Example:

```
#include<iostream.h>
#include<conio.h>
class ObjBuilt
{
 int x;
 double y;
 char z;
 public:
 ObjBuilt(int xx)
 {
  x=xx;
 }
 ObjBuilt(double yy)
 {
  y=yy;
 }
 ObjBuilt(char zz)
 {
  z=zz;
 }
 operator int()
```

```
{
   return x;
  }
  operator double()
  {
   return y;
  }
  operator char()
  {
  return z;
  }
};
void main()
{
  clrscr();
  ObjBuilt ob(10);
  int a=ob;
  cout<<a<<endl;
  ObjBuilt ob1(20.5);
  double aa=ob1;
  cout<<aa<<endl;
  ObjBuilt ob2('m');
  char aaa=ob2;
  cout<<aaa<<endl;
  ObjBuilt ob3(15);
  int ab=(int)ob;
  cout<<ab<<endl;
```

```
 getch();
}
```

Output:

10

20.5

m

10

In main(), the statement,

int a=ob;

converts the object “ob” to built-in data type “int” by invoking the overloaded operator function. The above conversion operator function can also be invoked explicitly as follows:

int a=(int)ob;

or

int a=int(ob);

The compiler invokes the appropriate conversion function. First, the compiler looks for an overloaded “=” operator. If does not find one, then it looks for a conversion operator function and involves the same implicitly for data conversion.

Conversion from one object to another object using conversion constructors:

```
#include<iostream.h>
#include<conio.h>
class objobjconv
{
 public:
 int a;
 char *name;
 char *dept;
```

```
public:
objobjconv()
{}
objobjconv(int aa,char *n,char *d)
{
 a=aa;
 name=n;
 dept=d;
 }
 void display()
 {
 cout<<a<<"\t"<<name<<"\t"<<dept<<endl;
 }
};
class objobjconv1
{

 int a_a;
 char *name_1,*dept_1;
 //int cgpa;
 public:
 objobjconv1()
 {}
 objobjconv1(objobjconv obj)
 {
 a_a=obj.a;
 name_1=obj.name;
```

```
  dept_1=obj.dept;
  }
  void display()
  {
  cout<<a_a<<"\t"<<name_1<<"\t"<<dept_1;
  }
};

void main()
{
clrscr();
objobjconv obj(20,"maria michael","cse");  //constructor used for conversion
obj.display();
objobjconv1 obj1;
obj1=obj;           //calling constructor function
obj1.display();
getch();
}
```

Output:

```
20      maria michael   cse
20      maria michael   cse
```

In main(), obj and obj1 are objects of different classes, obj is object of objobjconv class and obj1 is the object of objobjconv1 class. The statement,

Obj1=obj;

Converts the object obj into obj1 by invoking the constructor function.

Conversion from one object to another object using conversion operator function:

```
#include<iostream.h>
#include<conio.h>
class objobjoper
{
 public:
 int a;
 char *name;
 char *dept;
 public:
 objobjoper()
 {}
 objobjoper(int aa,char *n,char *d)
 {
  a=aa;
  name=n;
  dept=d;
 }
 void display()
 {
 cout<<a<<"\t"<<name<<"\t"<<dept<<endl;
 }
};
 class objobjoper1
 {

  int aa;
```

```
 char *name1,*dept1;
 int cgpa;
 public:
 objobjoper1()
 {}
 objobjoper1(int a_a,char *name_1,char *dept_1)
 {
 aa=a_a;
 name1=name_1;
 dept1=dept_1;

  }
 operator objobjoper()
 {
   return objobjoper(aa,name1,dept1);
   }
 void display()
 {
 cout<<aa<<"\t"<<name1<<"\t"<<dept1;
 }
};

void main()
{
clrscr();
objobjoper1 obj1(20,"michael","cse");
obj1.display();
```

```
objobjoper obj;
obj=obj1;
obj1.display();
getch();
}
```

Output:

```
20      maria michael   cse
20      maria michael   cse
```

In main(), the objects obj and obj1 are incompatible objects since the object obj is created for objobjoper class and obj1 belongs to class objobjoper1. In general, it is not possible to assign object of a class with object of another class, but the statement,

Obj=obj1;

Calls the conversion operator function to make this possible.