Class : BCA 3rd Semester **Course Code:** BCA-S3-03

Course Title: Object Oriented Programming Concepts in C++

Unit II

Constructor and Destructor

Constructor

It is a member function having same name as it's class and which is used to initialize the objects of that class type with a legel initial value. Constructor is automatically called when object is created.

Types of Constructor

Default Constructor-: A constructor that accepts no parameters is known as default constructor. If no constructor is defined then the compiler supplies a default constructor.

Circle :: Circle()

```
{
radius = 0;
}
```

Parameterized Constructor -: A constructor that receives arguments/parameters, is called parameterized constructor.

```
Circle :: Circle(double r)
{
radius = r;
}
```

Copy Constructor-: A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

```
Circle :: Circle(Circle&t)
{
radius =t.radius;
}
```

There can be multiple constructors of the same class, provided they have different signatures.

Destructor

A destructor is a member function having sane name as that of its class preceded by ~(tilde) sign and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object's scope is over.

```
~Circle() {}
```

Example : In the following program constructors, destructor and other member functions are defined inside class definitions. Since we are using multiple constructors in class so this example also illustrates the concept of constructor overloading

```
#include<iostream>
using namespacestd;

class Circle //specify a class
{
private :
double radius; //class data members
public:
Circle() //default constructor
        {
radius = 0;
        }
Circle(double r) //parameterized constructor
        {
radius = r;
        }
Circle(Circle&t) //copy constructor
        {
radius =t.radius;
        }
voidsetRadius(double r) //function to set data
        {
radius = r;
        }
doublegetArea()
        {
return 3.14 * radius * radius;
        }
        ~Circle() //destructor
        {}
};

int main()
{
    Circle c1; //defalut constructor invoked
    Circle c2(2.5); //parmeterized constructor invoked
    Circle c3(c2); //copy constructor invoked
cout<< c1.getArea()<<endl;
cout<< c2.getArea()<<endl;
cout<< c3.getArea()<<endl;
return 0;
}
```

Another way of Member initialization in constructors

The constructor for this class could be defined, as usual, as:

```
Circle :: Circle(double r)
{
radius = r;
}
```

It could also be defined using member initialization as:

```
Circle :: Circle(double r) : radius(r)
{}
```

Friend Functions

As we have seen in the previous sections, private and protected data or function members are normally only accessible by the code which is part of same class. However, situations may arise in which it is desirable to allow the explicit access to private members of class to other functions.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend. This is illustrated in the following code fragment:

```
#include <iostream>
using namespacestd;

class Rectangle
{
private :
int length;
int width;
public:

voidsetData(intlen,intwid)
        {
length =len;
width =wid;
        }

intgetArea()
        {
return length * width ;
        }

friend doublegetCost(Rectangle); //friend of class Rectangle
};

//friend function getCost can access private member of class
doublegetCost (Rectangle rect)
{
double cost;
cost =rect.length *rect.width * 5;
return cost;
}
```

```
int main ()
{
    Rectangle floor;
floor.setData(20,3);
cout<< "Expense "<<getCost(floor) <<endl;
return 0;
}
```

Output :

Expense 300

The getCost function is a friend of Rectangle. From within that function we have been able to access the members length and width, which are private members.

Friend Classes

One class memmber function can access the private and protected members of other class. We do it by declaring a class as friend of other class. This is illustrated in the following code fragment:

```
#include <iostream>
using namespacestd;

classCostCalculator;

class Rectangle
{
private :
int length;
int width;
public:
voidsetData(intlen,intwid)
        {
length =len;
width =wid;
        }

intgetArea()
        {
return length * width ;
        }

friend classCostCalculator; //friend of class Rectangle
};


//friend class costCalculator can access private member of class
Rectangle

classCostCalculator
```

```
{
public :
doublegetCost (Rectangle rect)
        {
double cost;
cost =rect.length *rect.width * 5;
return cost;
        }
};

int main ()
{
    Rectangle floor;
floor.setData(20,3);
CostCalculatorcalc;
cout<< "Expense "<<calc.getCost(floor) <<endl;
return 0;
}
```

Output :

Expense 300

Note : An empty declaration of class costCalculator at top is necessary.

Operator Overloading in C++

Operator overloading is giving new functionality to an existing operator. It means the behavior of operators when applied to objects of a class can be redefined. It is similar to overloading functions except the function name is replaced by the keyword operator followed by the operator's symbol. There are 5 operators that are forbidden to overload. They are :: . .* sizeof ?:

In the following code fragment, we will overload binary + operator for Complex number class object.

```
#include <iostream>
using namespacestd;

class Complex
{
private :
double real;
doubleimag;
public:
        Complex () {};
        Complex (double, double);
        Complex operator + (Complex);
void print();
};

Complex::Complex (double r, double i)
{
real = r;
imag = i;
```

```
}

Complex Complex::operator+ (Complex param)
{
    Complex temp;
temp.real = real +param.real;
temp.imag =imag +param.imag;
return (temp);
}

void Complex::print()
{
cout<< real<< " + i"<<imag<<endl;
}

int main ()
{
    Complex c1 (3.1, 1.5);
    Complex c2 (1.2, 2.2);
    Complex c3;

    c3 = c1 + c2; //use overloaded + operator

c1.print();
c2.print();
c3.print();
return 0;
}
```

Output :

3.1 + i1.5

1.2 + i2.2

4.3 + i3.7

In C++ we can cause an operator to invoke a member function by giving that member function a special name (of the form: operator<symbol>). Hence for the sum operation, the special name is: operator+. So, by naming the member function operator+ we can call the function by statement

c3 = c1 + c2

That is similiar to

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

Unary operators overloading in C++

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

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

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};

int main() {
   Distance D1(11, 10), D2(-5, 11);

   -D1;                     // apply negation
   D1.displayDistance();    // display D1

   -D2;                     // apply negation
   D2.displayDistance();    // display D2

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

F: -11 I:-10

F: 5 I:-11

Binary operators overloading in C++

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

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

class Box {
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box

public:

   double getVolume(void) {
      return length * breadth * height;
   }

   void setLength( double len ) {
      length = len;
   }

   void setBreadth( double bre ) {
      breadth = bre;
   }

   void setHeight( double hei ) {
      height = hei;
   }

   // Overload + operator to add two Box objects.
   Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
};
```

```
// Main function for the program
int main( ) {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here
```

```
   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
  Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

OVERLOADING FUNCTIONS

In many computer programming languages, each variable used in a function must have a unique name, but you learned earlier in this chapter that C++ allows you to employ an alias. Similarly, in many computer programming languages, each function used in a program must have a unique name. For example, if you want a function to display a value's square, you could create a function that squares integers, a function that squares floats, and a function that squares doubles. The code below shows three such functions, and each has a unique name.

```
void squareInteger(int x)
{
cout << "In function with integer parameter > " << x * x << endl;
}
void squareFloat(float x)
{
cout << "In function with float parameter > " << x * x << endl;
}
void squareDouble(double x)
{
cout << "In function with double parameter > " << x * x << endl;
}
```

So you don't have to use three names for functions that perform basically the same task, C++ allows you to reuse, or overload, function names. When you overload a function, you create multiple functions with the same name but with different parameter lists. For example, each function that squares a value can bear the name square(). If you want three versions of the function, you must still write three versions of square()—one that accepts an int, one that accepts a double, and one that accepts a float. C++ determines which of the three functions to call by reviewing the parameters submitted in your function call. Figure 6-38 shows three overloaded versions of square()and a main()function that uses them. Figure 6-39 shows the output.

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

void square(int x)
{
cout << "In function with integer parameter > " <<
x * x << endl;
}

void square(float x)
{
cout << "In function with float parameter > " <<
x * x << endl;
}

void square(double x)
{
cout << "In function with double parameter > " <<
x * x << endl;
}

int main()
{
int i = 5;
float f = 2.2f;
double d = 3.3;
square(i);
square(f);
```

```
square(d);
return 0;
}
```

Inline Functions

C++ inline function is powerful concept that is commonly used with classes. If a function is inline , the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

Following is an example, which makes use of inline function to return max of two numbers:

```
#include <iostream>

using namespace std;

inline int Max(int x, int y) {
   return (x > y)? x : y;
}

// Main function for the program
int main( ) {

   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;

   return 0;
}
When the above code is compiled and executed, it produces the
following result:
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```