UNIT 2

Constructors and Destructors In C++

Constructors:

What is the use of Constructor

The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

General Syntax of Constructor

Constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

<class name> { arguments};

The default constructor for a class X has the form

X::X()
In the above example the arguments is optional.
The constructor is automatically invoked when an object is created.

The various types of constructors are

- Default constructors
- Parameterized constructors
- Copy constructors

Default Constructor:

This constructor has no arguments in it. Default Constructor is also called as *no argument constructor*.

For example:

```
Class Exforsys
{
   private:
      int a,b;
   public:
      Exforsys(); //default
Constructor
      ...
};

Exforsys :: Exforsys()
```

```
{
   a=0;
   b=0;
}
```

Parameterized Constructor:

A parameterized constructor is just one that has parameters specified in it.

Example:

```
class Exforsys
{
   private:
      int a,b;
   public:
      Exforsys(int,int);// Parameterized constructor
      ...
};

Exforsys :: Exforsys(int x, int y)
{
   a=x;
   b=y;
}
```

Copy constructor;

One of the more important forms of an overloaded constructor is the copy constructor. The purpose of th copy constructor is to initialize a new object with data copied from another object of the same class.

For example to invoke a copy constructor the programmer writes:

```
Exforsys e3(e2);
or
Exforsys e3=e2;
```

Both the above formats can be used to invoke a copy constructor.

For Example:

```
#include <iostream.h>
class Exforsys()
{
   private:
      int a;
   public:
      Exforsys()
      { }
      Exforsys(int w)
   {
      a=w;
   }
   Exforsys(Exforsys& e)
   {
      a=e.a;
      cout<<" Example of Copy
Constructor";
   }
   void result()
   {
      cout<< a;
   }
};

void main()
{
   Exforsys e1(50);
   Exforsys e3(e1);
   cout<< "\ne3=";e3.result();
}
```

In the above the copy constructor takes one argument an object of type Exforsys which is passed by reference. The output of the above program is

Example of Copy Constructor
e3=50

Some important points about constructors:

- A constructor takes the same name as the class name.
- The programmer cannot declare a constructor as virtual or static, nor can the programmer declare a constructor as const, volatile, or const volatile.

- No return type is specified for a constructor.
- The constructor must be defined in the public. The constructor must be a public member.
- Overloading of constructors is possible.

Destructors

What is the use of Destructors?

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically called when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

The above is the general syntax of a destructor. In the above, the symbol tilda ~ represents a destructor which precedes the name of the class.

Some important points about destructors:

- Destructors take the same name as the class name.
- Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
- The Destructor does not take any argument which means that destructors cannot be overloaded.
- No return type is specified for destructors.

For example:

```
class Exforsys
{
   private:
      ……….
   public:
      Exforsys()
      { }
      ~ Exforsys()
      { }
}
```

Operator Overloading

Operator overloading is a very important feature of Object Oriented Programming. It is because by using this facility programmer would be able to create new definitions to existing operators. In other words a single operator can perform several functions as desired by programmers.

Operators can be broadly classified into:

- Unary Operators
- Binary Operators

Unary Operators:

As the name implies takes operate on only one operand. Some unary operators are namely

++ - Increment operator

-- - Decrement Operator

! - Not operator

• unary minus.

Binary Operators:

The arithmetic operators, comparison operators, and arithmetic assignment operators come under this category.

Both the above classification of operators can be overloaded. So let us see in detail each of this.

Operator Overloading – Unary operators

As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword **operator**.

The general syntax for defining an operator overloading is as follows:

```
return_type classname :: operator operator symbol(argument)
{
……..
statements;
}
```

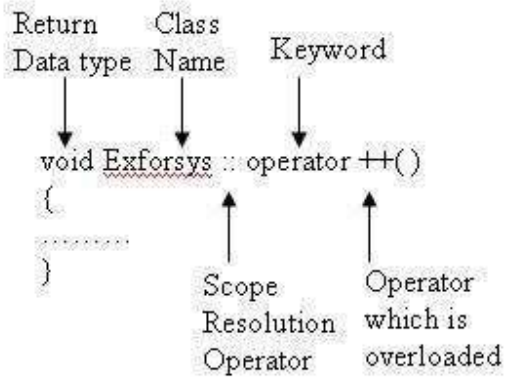
Thus the above clearly specifies that operator overloading is defined as a member function by making use of the keyword operator.

In the above:

- return_type – is the data type returned by the function
- class name - is the name of the class
- operator – is the keyword
- operator symbol – is the symbol of the operator which is being overloaded or defined for new functionality
- **::** - is the scope resolution operator which is used to use the function definition outside the class.

For example

Suppose we have a class say Exforsys and if the programmer wants to define a operator overloading for unary operator say ++, the function is defined as



Inside the class Exforsys the data type that is returned by the overloaded operator is defined as

```
class Exforsys
{
   private:
   …….
   public:
   void operator ++( );
   ……
};
```

The important steps involved in defining an operator overloading in case of unary operators are namely:

- Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function.
- If the function is a member function then the number of arguments taken by the operator member function is none.
- If the function defined for the operator overloading is a friend function then it takes one argument.

Now let us see how to use this overloaded operator member function in the program

```
#include <iostream.h>
class Exforsys
{
   private:
   int x;
   public:
   Exforsys( ) { x=0; }       //Constructor
   void display();
   void Exforsys ++( );          //overload unary ++
};

void Exforsys :: display()
{
   cout<<"\nValue of x is: " << x;
}

void Exforsys :: operator ++( ) //Operator Overloading for operator ++
                                        defined
{
   ++x;
}

void main( )
{
   Exforsys e1,e2;         //Object e1 and e2 created
   cout<<"Before Increment"
   cout <<"\nObject e1: "<<e1.display();
   cout <<"\nObject e2: "<<e2.display();
   ++e1;  //Operator overloading applied
   ++e2;
   cout<<"\n After Increment"
   cout <<"\nObject e1: "<<e1.display();
   cout <<"\nObject e2: "<<e2.display();
}
```

The output of the above program is:

crackptu

Before Increment
Object e1:
Value of x is: 0
Object e1:
Value of x is: 0
Before Increment
Object e1:
Value of x is: 1
Object e1:
Value of x is: 1

In the above example we have created 2 objects e1 and e2 f class Exforsys. The operator ++ is overloaded and the function is defined outside the class Exforsys.

When the program starts the constructor Exforsys of the class Exforsys initialize the values as zero and so when the values are displayed for the objects e1 and e2 it is displayed as zero. When the object ++e1 and ++e2 is called the operator overloading function gets applied and thus value of x gets incremented for each object separately. So now when the values are displayed for objects e1 and e2 it is incremented once each and gets printed as one for each object e1 and e2.

Operator Overloading – Binary Operators

Binary operators, when overloaded, are given new functionality. The function defined for binary operator overloading, as with unary operator overloading, can be member function or friend function.

The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one (see below example). When the function defined for the binary operator overloading is a friend function, then it uses two arguments.

Binary operator overloading, as in unary operator overloading, is performed using a keyword operator.

Binary operator overloading example:

```
#include <iostream.h>
class Exforsys
{
private:
int x;
int y;
```

```
public:
Exforsys()                  //Constructor
{ x=0; y=0; }

void getvalue( )            //Member Function for Inputting Values
{
cout << "\n Enter value for x: ";
cin >> x;
cout << "\n Enter value for y: ";
cin>> y;
}

void displayvalue( )        //Member Function for Outputting Values
{
cout <<"value of x is: " << x <<"; value of y is: "<<y
}



Exforsys operator +(Exforsys);
};
                                                    )
Exforsys Exforsys :: operator + (Exforsys e2
//Binary operator overloading for + operator defined
{
int x1 = x+ e2.x;
int y1 = y+ e2.y;
return Exforsys(x1,y1);
}

void main( )
{
Exforsys e1,e2,e3;          //Objects e1, e2, e3 created
cout<<\n"Enter value for Object e1:";
e1.getvalue( );
cout<<\n"Enter value for Object e2:";
e2.getvalue( );
e3= e1+ e2;                 //Binary Overloaded operator used
cout<< "\nValue of e1 is:"<<e1.displayvalue();
cout<< "\nValue of e2 is:"<<e2.displayvalue();
cout<< "\nValue of e3 is:"<<e3.displayvalue();
}
```


crackptu

The output of the above program is:

```
Enter value for Object e1:
Enter value for x: 10
Enter value for y: 20
Enter value for Object e2:
Enter value for x: 30
Enter value for y: 40
Value of e1 is: value of x is: 10; value of y is: 20
Value of e2 is: value of x is: 30; value of y is: 40
Value of e3 is: value of x is: 40; value of y is: 60
```

In the above example, the class Exforsys has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator '+' is declared as a member function inside the class Exforsys. The definition is performed outside the class Exforsys by using the scope resolution operator and the keyword operator.

The important aspect is the statement:

```
e3= e1 + e2;
```

The binary overloaded operator '+' is used. In this statement, the argument on the left side of the operator '+', e1, is the object of the class Exforsys in which the binary overloaded operator '+' is a member function. The right side of the operator '+' is e2. This is passed as an argument to the operator '+' . Since the object e2 is passed as argument to the operator'+' inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y. The return value is of type class Exforsys as defined by the above example.

There are important things to consider in operator overloading with C++ programming language. Operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary.

Some operators cannot be overloaded:

- Scope resolution operator denoted by ::
- Member access operator or the dot operator denoted by .
- Conditional operator denoted by ?:
- Pointer to member operator denoted by .*

Operator Overloading through friend functions

```
// Using friend functions to
// overload addition and subtarction
// operators
#include <iostream.h>

class myclass
  {
int a;
int b;

public:
myclass(){}
myclass(int x,int y){a=x;b=y;}
void show()
   {
cout<<a<<endl<<b<<endl;
   }

// these are friend operator functions
// NOTE: Both the operans will be be
// passed explicitely.
// operand to the left of the operator
// will be passed as the first argument
// and operand to the right as the second
// argument
friend myclass operator+(myclass,myclass)
friend myclass operator-(myclass,myclass);

  };

myclass operator+(myclass ob1,myclass ob2)
  {
myclass temp;
temp.a = ob1.a + ob2.a;
temp.b = ob1.b + ob2.b;

return temp;
  }

myclass operator-(myclass ob1,myclass ob2)
  {
myclass temp;
temp.a = ob1.a - ob2.a;
temp.b = ob1.b - ob2.b;

return temp;
```

```
}

void main()
  {
myclass a(10,20);
myclass b(100,200);

a=a+b;
a.show();
  }
```

Overloading the Assignment Operator (=)

We know that if we want objects of a class to be operated by common operators then we need to *overload them*. But there is one operator whose operation is automatically crested by C++ for every class we define, it is the assignment operator '='.

Actually we have been using similar statements like the one below previously

ob1=ob2;

where ob1 and ob2 are objects of a class.

This is because even if we don't overload the '=' operator, the above statement is valid.

because C++ automatically creates a default assignment operator. The default operator created, does a member-by-member copy, but if we want to do something specific we may overload it.

The simple program below illustrates how it can be done. Here we are defining two similar classes, one with the default assignment operator (created automatically) and the other with the overloaded one. Notice how we could control the way assignments are done in that case.

```
// Program to illustrate the
// overloading of assignment
// operator '='
#include <iostream.h>

// class not overloading the
// assignment operator
class myclass
  {
int a;
int b;
```

```
public:
myclass(int, int);
void show();
  };

myclass::myclass(int x,int y)
  {
a=x;
b=y;
  }

void myclass::show()
  {
cout<<a<<endl<<b<<endl;
  }

// class having overloaded
// assignment operator
class myclass2
  {
int a;
int b;

public:
myclass2(int, int);
void show();

myclass2 operator=(myclass2);
  };

myclass2 myclass2::operator=(myclass2 ob)
  {
// -- do something specific—
// this is just to illustrate
// that when overloading '='
// we can define our own way
// of assignment
b=ob.b;

return *this;
  };

myclass2::myclass2(int x,int y)
  {
a=x;
```

```
b=y;
  }

void myclass2::show()
  {
cout<<a<<endl<<b<<endl;
  }

// main
void main()
  {
myclass ob(10,11);
myclass ob2(20,21);

myclass2 ob3(100,110);
myclass2 ob4(200,210);

// does a member-by-member copy
// '=' operator is not overloaded
ob=ob2;
ob.show();


// does specific assignment as
// defined in the overloaded
// operator definition
ob3=ob4;
ob3.show();
  }
```

Type Conversions in C++

What is Type Conversion

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.

How to achieve this

There are two ways of achieving the type conversion namely:

Automatic Conversion otherwise called as **Implicit Conversion**
Type casting otherwise called as **Explicit Conversion**

Let us see each of these in detail:

Automatic Conversion otherwise called as Implicit Conversion

This is not done by any conversions or operators. In other words value gets automatically converted to the specific type in which it is assigned.

Let us see this with an example:

```
#include <iostream.h>
void main()
{
short x=6000;
int y;
y=x;
}
```

In the above example the data type short namely variable x is converted to int and is assigned to the integer variable y.

So as above it is possible to convert short to int, int to float and so on.

Type casting otherwise called as Explicit Conversion

Explicit conversion can be done using type cast operator and the general syntax for doing this is

datatype (expression);

Here in the above datatype is the type which the programmer wants the expression to gets changed as

In C++ the type casting can be done in either of the two ways mentioned below namely:

- C-style casting
- C++-style casting

The C-style casting takes the syntax as

(type) expression

The C++-style casting takes the syntax as

type (expression)

Let us see the concept of type casting in C++ with a small example:

```
#include <iostream.h>
void main()
{
int a;
float b,c;
cout<< “Enter the value of a:”;
cin>>a;
cout<< “n Enter the value of b:”;
cin>>b;
c = float(a)+b;
cout<<”n The value of c is:”<<c;
}
```

The output of the above program is

Enter the value of a: 10
Enter the value of b: 12.5
The value of c is: 22.5

In the above program ‘a’ is declared as integer and b and c are declared as float. In the type conversion statement namely

c = float(a)+b;

The variable a of type integer is converted into float type and so the value 10 is converted as 10.0 and then is added with the float variable b with value 12.5 giving a resultant float variable c with value as 22.5

Explicit Constructors

The keyword explicit is a Function Specifier.” The explicit specifier applies only to constructors. Any time a constructor requires only one argument either of the following can be used to initialize the object. The reason for this is that whenever a constructor is created that takes one argument, it also implicitly creates a conversion from the type of that argument to the type of the class. A constructor specified as explicit will be used only when an initialization uses the normal constructor syntax, Data (x). No automatic conversion will take place and Data = x will not be allowed. Thus, an explicit constructor creates a “nonconverting constructor.”

Example:

```
class Data
{
```

```
explicit Data(float x); // Explicit constructor
{ }

};
```

Implicit Constructors

If a constructor is not stated as explicit, then it is by default an implicit constructor.