SATHYABAMA UNIVERSITY FACULTY OF ELECTRICAL AND ELECTRONICS

OBJECT ORIENTED PROGRAMMING (SCS1202)

UNIT 3 CONSTRUCTORS AND OVERLOADING

Default constructors-Parameterized constructors-Constructor overloading-Copy constructors-new, delete operators-"this" pointer-friend classes and friend functions-Function overloading- Unary Operator overloading -Binary Operator overloading.

CONSTRUCTOR :-

A constructor is special member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

CHARACTERISTICS:-

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.

A constructor is declared and defined as follows.

```
class integer
{
   int m, n ;
   public :
        integer  (void ) ;
};
integer : : integer(void )
{
```

```
         m=o ;
         n=o ;
        cout <<"m="<<m<<" n="<<n ;
}
void main( )
{
     integer int ;
}
```

Output:-

```
        m=0 n=0
```

<u>PARAMETERIZED CONSTRUCTORS:-</u>

C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterized constructors. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways.

1. By calling the constructor **explicitly** integer int1=integer(0,100);
2. By calling the constructor **implicitly** integer int1(0,100);

Example:-

```
#include<iostream.h>
class sample
{
                int m, n ;
        public:
                sample(  int, int ) ;
 } ;
sample : : sample (int  x,  int  y )
{
        m=x ;
        n=y ;
       cout<<m<<n ;
}
```

```
        void  main  ( )
        {
                 sample S(25,50);
        }
```

Output:-

```
              25       50
```

<u>MULTIPLE CONSTRUCTOR:-</u>

```
class  integer
{
    int  m, n ;
    public :
          integer( )
          {
             m = 0 ;
              n = 0 ;
               cout<<m<<n<<endl ;
           }
           integer  ( int  a,  int  b )
           {
               m = a ;
               n =  b ;
               cout<<m<<n ;
            }
integer(integer &i )
    {
          m  =  i. m ;
          n  =  i.  n ;
    }
};
    void  main( )
    {
        integer  int1 ;
        integer  int2( 10, 20)
        integer int3(int1) ;
    }
```

Output:-

```
        0 0
        10 20
        10 20
```

<u>COPY CONSTRUCTOR:-</u>

A constructor can accept a reference to its own class as a parameter.

```
                    class  A
                    {
                         ………….
                         …………….
                         ……………..
                         public :
                            A(A&) ;
                     } ;
```

is valid. In such cases, the constructor is called copy constructor.

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in the following program.

<u>Example:-</u>

```
#include<iostream.h>
class  code
```

```
     {
          int  id ;
          public :
               code( )
               {
               }
               code(int a)
               {
                  id = a;
               }
               code(code  &x)
               {
                  id = x. id ;
               }
               void  display( void )
               {
                   cout<<id ;
               }
} ;
int main ( )
{
     code A ( 100 ) ;
     code  B  (A ) ;
     code  C = A ;
     code  D ;
     D = A :
      cout<<"/n  id  of  A : " ;
      A. display  ( ) ;
      cout<<"/n  id  of  B : " ;
      B.  display  ( ) ;
      cout<<"/n  id  of  C : " ;
      C.  display  ( ) ;
      cout<<"/n  id  of  D : " ;
      D.  display  ( ) ;
      return 0 :
}
```

Output:-

```
        id of A:100
        id of B:100
        id of C:100
        id of D:100
```

<u>DESTRUCTOR:-</u>

It is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

```
~integer( )
   {
  -----------
   }
```

<u>Example:-</u>

```
#include<iostream.h>
 int  count = 0 ;
 class  alpha
  {
      alpha  ( )
        {
                cout<< " object created "
        }
        ~ alpha  ( )
              {
                    cout<< " object  destroyed  ;
              }
      int  main( )
      {
         alpha  A1 ;                   }
```

Output:- object created

```
object destroyed
```

SCOPE RESOLUTION OPERATOR:-

Blocks and scopes can be used in construction programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block. In C , the global version of a variable cannot be accessed from within the inner block. In C++ resolves this problem by introducing a new operator :: is called the scope resolution operator. This can be used to uncover a hidden variable. It takes the following form.

:: variable-name

Program for scope resolution operator:-

```
#include<iostream.h>
int m=10;// global m
void main()
{
        int m=20;//m redeclared, local to main
        {
                int k=m;
                int m=30;//m declared again, local to inner block
                cout<<"we are inner block \n";
                cout<<"k="<<k<<"\n";
                cout<<"m="<<m<<"\n";
```

```
                    cout<<"::m"<<:: m<<"\n";
            }
        cout<<"\n we are in outer block \n";
        cout<<"m="<<m<<"\n";
        cout<<"::m"<<::m<<"\n";
        }
```

Output:-

```
        we are in inner block
        k=20
        m=30
        ::m=20
        we are in outer block
        m=20
        ::m=10
```

<u>MEMORY MANAGEMENT OPERATORS</u>:-

C++ supports two unary operators new and delete that perform the task of allocating and freeing the memory. An object can be created by using new, and destroyed by using delete. C++ uses a unique keyword called this to represent an object that invokes a member function. This is a pointer that points to the object for which this function was called.

```
                int   *p  =  new  int ;
```

We can also initialize the memory using the new operator.

```
            pointer-variable = new data-type(value);
            int   *p = new int (25);
           float   *q  =  new  float  ( 7.5 );
```

DELETE:-

When a data object is no longer needed it is destroyed to release the memory space for reuse.

```
delete pointer-variable ;
delete p ;
delete q ;
```

MANIPULATORS:-

Manipulators are operators that are used to format the data display. The most commonly used manipulators are end l and set w.

endl -->

when used in an output statement , causes a linefeed to be inserted. It has the same effect as using the newline character " \ n ".

Example:-

```
……………………..
……………………..
………………………
cout  <<  " m = "<<m<<end l << " n = "<<n<<end l <<
                    "p = "<<p<<endl ;
```

Where

m = 2567, n = 14, and p = 175

The output will appear like this.

m=

2	5	9	7

n=

1	4

p=

1	7	5

This form is not the ideal output. It should appear like this

```
m=  2 5  9   7
 n=          1  4
 p=      1  7  5
```

The numbers are right—justified. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right—justified.

The **setw** manipulator does this job.

Example:-

```
 cout << setw(5) <<sum << end l ;
```

setw(5) specifies a field width 5 for printing the value of the variable sum. The value is right justified within the field as shown below.

		3	4	5

PROGRAM:-

```
#include <iostream .h>
#include <iomanip.h>
int  main( )
{
int  Basic = 950 ;
int  Allowance = 95 ;
int  Total = 1045 ;
cout << setw(10)<< “Basic” <<setw (10) << Basic << endl
         << setw(10)<< “Allowance “ << setw(10)<<Allowance<<endl
         << setw(10)<<”Total”<<setw(10)<< Total << endl ;
 return 0 :
  }
```

Output:-

```
     BASIC       950
 ALLOWANCE        95
     TOTAL      1045
```

TYPE CAST OPERATOR:-

C++ permits explicit type conversion of variables or expressions using the type cast operator.

(type-name) expression -→ c

type-name (expression) -→ c ++

Example :-

Average = sum /(float)i ;

Average = sum /float(i) ;

FRIEND CLASSES AND FRIEND FUNCTIONS

FRIEND FUNCTION

A function that has access to the private member of the class but is not itself a member of the class is called friend functions.The general form is

friend data_type function_name( );

Friend function is preceded by the keyword 'friend'.

Consider a situation of operating on objects of two different classes, in such situation, friend function can be used to bridge the two classes.

Example:

}

```
#include<iostream.h>
class result;   //advance declaration
class test
{
 int n1,n2;
public:
void getmark( )
{
n1=45;
n2=78;
}
friend void display(  test, result);   // friend function declaration
};
class result
{
 int tot;
public:
friend void display(test,result);      // friend function declaration
};
void display( test t, result r)
{
r.tot= t.n1+t.n2;       // private member of test class accessed using t object
cout<<r. tot;
}
void main( )
{
          test t;
          result r;
          t.getmark( );
          display( t, r);
}
```

EXPLANATION:

- The program contains two classes test and result.
- The display function is friend function to access the private data members of both the classes.
- Declaration of the friend function can be placed either in the private or public section of the class.
- An object of each class has been passed as an argument to the function display ( ).
- It can access the private members of both classes through these arguments.
- The advanced declaration of the class result; in the beginning of the program is necessary, since a class cannot be referred until it has been declared before the class test.

USES OF FRIEND FUNCTION:

1. Function operating on objects of two different classes.
2. Friend function can be used to increase the versatility of overloaded operators.

FRIEND CLASSES

Definition:

All the functions of another class to manipulate or use the private members of the friend class.

Example:

```
#include<iostream.h>
class two;
class one
{
 private:
        int x, y;
public:
```

```
          void setdata(int a, int b)
          {
                    x=a; y=b;
          }
friend class two;
};
class two
{
          int z;
          public:
          void  addone(one obj)
          {
                    z= obj. x + obj . y;
          }
          void display(one obj )
          {         cout<<”X is: “<<obj.x<<”Y is:”<<obj.y;
                    cout<< “sum of one class data members”<< z;
          }
};
void main( )
{
 one obj1;
 two obj2;
obj1. setdata( 45,65);
obj2. addone(obj1);
obj2. display( obj1);
}
```

Run

```
X is: 45
Y is: 65
sum of one class data members: 110
```

EXPLANATION:

- Class two is friend to class one, so all the private members of class one can be accessed in member function of class two using class one object.
- Here addone () and display () functions using the private data members of class one. So both the functions are having argument as object of class one.

THIS POINTER:-

C++ uses a unique keyword called this to represent an object that invokes a member function. This is a pointer that points to the object for which this function was called.

For example:-

The function call A. max ( ) set the pointer this to the address of the object A. One important application of the pointer this is to return the object it points to.

return * this ;

```
#include <iostream.h>
class  person
{
         float  age ;
         public :
                   person ( float  a )
                   {
                       age = a ;
                   }
                   person &  person  : :  greater  ( person  &  x )
                   {
                           if     ( x. age > = age )
```

```
                    return x ;
               else
                    return * this ;
             }
             void  display  ( void )
             {
                   cout <<"age ="<< age ;
              }
          } ;
           void  main  ( )
          {      person  P1  (37.50 ), P2 ( 29.0), P3 (40.25);
                  person  P = P1. Greater  (P3);
                  cout<<" Elder  person  is " ;
                  P. display ( );
                P = P1. Greater  (P2) ;
                  cout <<"Elder  person  is :
                  P. display ( ) ;
        }
```

Output:-

```
Elder  person  is
Age : 40.25
Elder  person  is       Age : 37.5
```

<u>OVERLOADING IN C++:-</u>

Function overloading, Types of function overloading, Operator overloading, Overloading unary operator and Overloading binary operator. Templates, Generic functions and Generic classes.

<u>FUNCTION:-</u>

A function is a set of instructions that are used to perform specified tasks which repeatedly occurs in the main program.

By using function we can divide complex tasks into manageable tasks. The function can also help to avoid duplication of work.

Call by value:-

This method copies the values of actual parameters into the formal parameters of the function, because formal arguments are photocopy of actual arguments. Changes made in the formal arguments are local to the block of the called functions. Once control returns back to the calling function the changes made disappear.

Example:-

```
#include<iostream.h>
class call
{
        int a;
        public:
                void getdata (int p)
                {
                        a=p;
                        cout<< a;
                }

} ;
void main ( )
{
        call c;
        int x;
        c.getdata (3) }
```

Return by value:-

```
#include<iostream.h>
class call
{
    int a;
    public:
          int getdata (int   p)
```

```
            {
                 a=p;
                return p;
            }
};
void main ( )
{
          call c
           int x;
           x=c.getdata (3);
           cout<<x;
 }
```

Call by reference:-

Call by reference is another way of passing parameters to the function. Here, the address of arguments are copied into the parameters inside the function, the address is used to access the actual arguments used in the call. Hence changes made in the arguments are permanent.

Example:-

```
#include<iostream.h>
class sample
  {
         int t;
         public:
                 void swap (int & a, int & b)
                    {
                        t=a;
                        a=b;
                        b=t;
                    }
   } ;
```

```
    void main ( )
       {
             sample s;
             int i=5, j=10, k;
             k = s. swap (i, j);
             cout<<k;
         }
```

i, j → it will pass the value of i&j .

& a, & b→it will take the address of a & b [i.e., x, y]

*a, * b-→it takes the value of a & b.

```
#include<iostream.h>
class sample
{
         int t ;
         public:
                        void swap (int *a, int * b)
                   {
                         t = * a;
                        *a= * b;
                        *b= t;
                   }
} ;
void main ( )
{
         sample s;
         int x, y;
         cin>>x>>y;
         s.swap (&x, &y)
)
```

&x, &y, pass address value

*a, *b→ takes the value of x & y

<u>Return by reference:-</u>

A function can also return a reference.

```
#include<iostream.h>
class sample
{
        public:
             int &max (int &x, int &y)
            {
                        if(x > y)
                        return x;
                        else
                        return y;
             }
};
void main()
{
        sample S,T;
        T=S.max(5,2)
        cout<<"greater"<<T;
}
```

Output:-

```
greater 5
```

Since the return type of max ( ) is int &, the function returns reference to x or y (and not the values). Then a function call such as max (a, b) will yield a reference to either a or b depending on their values.

Max (a, b) is legal and assigns-1 to a if it is larger, otherwise -1 to b.

<u>FUNCTION OVERLOADING:-</u>

We can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in oop. Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

Example:-

Declarations:-

1. int add (int a, int b);
2. int add (int a, int b, int c);
3. double add (double x, double y);
4. double add (int p, double q);
5. double add (double p, int q);

Function calls:-

```
cout << add (0.75, 5);     // uses 5
cout << add (5, 10);       // 1
cout << add (15, 10.0); // 4
cout << add (12.5, 7.5); // 3
cout << add (5, 10.15); //2
```

```
#include<iostream.h>
class funoverloading
  {
        int a, b, c;
        public:
            void add ( )
              {
                    cin>>a>>b;
                    c = a + b;
                    cout << c;
               }
             int add (int a, int b);
             {
                  c = a + b;
```

```
                    return c;
                }
                void add (int a)
                {
                       cin>>b;
                       c = a + b;
                       cout<<c;
                 }
   }
}
void main ( )
{
    funoverloading F;
    int x;
    F. add ( );
    x = F. add (10, 5);
    cout<<x;
    F. add (5);
}
```

Output:-

```
    5       3
    c =   8
    c = 15
    5
    c = 10
```

<u>OPERATOR OVERLOADING:-</u>

C ++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

The process of overloading involves the following steps:

- Create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op()in the public part of the class
- It may be either a member function or a friend function
- Define the operator function to implement the required operations.

Syntax:-

```
    returntype classname:: operator op (argument list)
        {
              Function body
        }.
```

Example:-

```
      void space:: operator-( )
      {
            x=-x;
      }
```

<u>OVERLOADING UNARY OPERATORS:-</u>

Let us consider the unary minus operator. A minus operator when used as a unary takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here 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 variable. The unary minus when applied to an object should change the sign of each of its data items.

```
#include<iostream.h>
{
        int x;
        int y;
        int z;
        public:
                void getdata(int a, int b, int c);
                void display(void);
                void operator-(); //overload unary minus
};
void space::getdata(int a, int b, int c)
{
        x=a;
        y=b;
        z=c;
}
void  space::display(void)
{
        cout<<x<<" ";
        cout<<y<<" ";
        cout<<z<<" ";
}
void space::operator-()
{
        x=-x;
        y=-y;
        z=-z;
}
int main()
{
        space S;
        S.getdata(10,-20,30);
        cout<<"S=";
        S.display();
        - S;
        cout<<"S=";
        S.display();
        return 0;
}
```

Output:-

```
        S= 10 -20 30
        S=  -10 20 -30
```

Note:

The function operator-() takes no argument. Then, what does this operator function do? It changes the sign of data members of the object S. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

<u>OVERLOADING BINARY OPERATORS:-</u>

The same mechanism which is used in overloading unary operator can be used to overload a binary operator.

```
# include<iostream.h>
class complex
{
        float x;
        float y;
        public:
                complex()
                {
                }
                complex(float real, float imag)
                {
                        x=real;
                        y=imag;
```

```
                    }
                    complex operator+(complex);
                    void display(void);
};
complex complex::operator+(complex c)
{
          complex temp;
          temp.x=x+c.x;
          temp.y=y+c.x;
          return(temp);
}
void complex::display(void)
{
          cout<<x<<”j”<<y<<”\n”;
}
int main()
{
          complex C1,C2,C3;
          C1=complex(2.5,3.5)
          C2=complex(1.6,2.7)
          C3= C1 + C2;
          cout<<”C1 = ”;
          C1.display();
          cout<<”C2 = ”;
          C2.display();
          cout<<”C3 = ”;
          C3.display();
          return 0;
}
```
Output:-

```
          C1=2.5 +j3.5
```

$C2=1.6 + j2.7$

$C3= 4.1 +j6.2$