DEPARTMENT OF COMMERCE (CA)

OBJECT ORIENTED PROGRAMMING WITH C++ (SEMESTER-IV)

II-B.COM (CA) **SUBJECT CODE - 18BCA42C**

UNIT – IV

Constructor and destructors: Introduction – copy constructors. Dynamic Constructors – destructors operator overloading: Definition of operator overloading – Overloading binary operators using friends – manipulation of string using operators – Rules for overloading operators – Types conversion.

Constructors :

- Constructor is a special member function that has the same name of the class and provides the way to initialize the objects of the class.
- The Constructor is invoked implicitly whenever an object of the class is created.
- It is called as Constructor because it reserves memory for newly created object and initialize members of an object with value.
- Example:

```
//  Class with a Constructor
class  sample
{
        int m,n;
        public :
                sample( );              //….. constuctor declared…..
                 ----
                -----
};
sample  : : sample( )           // ….constructor defined….
{
        m=0;
        n=0;
}
// ….main   function ….
int main( )
{
        sample  s1.

}
```

This statement creates the object ‘s1’ of the type ‘sample’ and also initializes its data members ‘m’ and ‘n’ to zero.

Types Of Constructors

- Default Constructor
- User defined Constructor(parameterized Constructor)
- Copy Constructor.

Characteristics of Constructor function:

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- The name must be the same as its class name
- They cannot return values (not even void)
- They can not be Static or Virtual.
- Each time an object of a class is defined, its Constructor is applied to it implicitly.

Default Constuctor

Default Constructor is a constructor that accepts no parameters. Here no processing is needed and the Constructor simply reserves memory by default.

Example:

```
//    example for    default Constructor
# include < iostream.h>
class student
{
        int rollno;
        char name[30];
        float height,weight;

        public:
                student ( );                // constructor
                void display ( );

};

student ::student ( )
{

        name [0]='\0';
        rollno=0;
        height=0;
        weight=0;
}
```

```
void  student : :  display ( )
{
        cout << "Name "  <<  name << endl;
        cout << "Rollno"  <<  rollno << endl;
        cout << "Height "  << height << endl;
        cout << "Weight "  << weight << endl;
}

int main ( )
{
        student s;
        cout <<"The values from default Constructor are :" << endl'
        s.display( );
        return 0;
  }
```

output

```
The values from  default Constructor are :
Name=
Rollno=0
Height=0
Weight=0
```

Parameterized constructors:

The parameterized Constructor are Constructor with arguments in order to pass different values to initialize the data members from the main ( ).

Example:

```
//  Mark pattern  for  UG  and  PG  using  parameterized Constructor

# include <iostream.h>
class   mark
{
        int internal,external;
        public:
                mark(int,int);          //    constructor  declared.

                Void display(void)
                {
                        cout <<"Internal  mark ="<< internal <<endl;
                        cout <<"External mark="<<external << endl;
```

```
                }
};


mark :: mark( int i,int e)                  //  constructor  defined.
{
        internal =i
        external=e;

}

int  main ( )
{
        mark ug(0,100) ;
        mark pg(25,75);

        cout <<"   UG  pattern    "<<endl;
        ug.display ( );
        cout <<" PG  pattern"<<endl;
        pg.display( );
        return 0;
}
```

Constructors with default arguments:

- We can define Constructor with default arguments
- In the above example , the Constuctor integer can be declared as follows,

integer (int a, int b=10)

The statement integer eg(20) assigns the value 20 to ‘a’ and the value 10 to ‘b’ (by default)

Difference between default Constructor and default argument:

- When the Constructor is called with no argument , it becomes default Constructor.
- But a default argument Constructor call be called with either one argument or no argument

Copy constructors:

A copy Constructor takes a reference to an object of the same class as itself as an argument. Example:

```
//  Generation oof Fibonacci series using  copy Constructor
# include <iostream.h>
class  fibonacci
{
        int  f0,f1,fib;
        public:
                fibonacci ( );                  // constructor
                fibonacci(fibonacci &  ptr);    // copy Constructor
                void  increment ( );
                viod  display( );
};
fibonacci  :  :  fibonacci ( )
{
        f0=0;
        f1=1;
        fib=f0+f1;
}

fibonacci  : : fibonacci ( fibonacci  & ptr)
{
        f0=ptr  . f0 ;
        f1=ptr  . f1 ;
        fib = ptr .  fib ;
}

 void  fibonacci  ::  increment ( )
{
        f0= f1;
        f1= fib;
        fib=f0 +f1;
}

void  fibonacci  : : display  ( )
{
        cout  <<  fib << endl;
}
int  main ( )
{
        int  n;
        fibonacci  number;
```

```
        cout <<"How  many   numbers do you want  to create"<<endl;
        cin  >> n;
        fibonacci   gen(number);
        for(int i=0;i<n;++i)
        {
                gen.display    ( );
                gen.increment ( );
        }

}
```

Dynamic Constructors

Allocation of memory to objects at the time of Constructon is known as dynamic Construction of objects. The memory is allocated with the help of new operator.

Example :

```
//      Program to  join   strings
# include <iostream.h>
# include  <string.h>

class  string
{
        char   * name;
        int      length;

        public:
                string( )                    // constructor –1
                {
                        length=0;
                        name=new char [length +1];
                //     one additional character to store  '\0'
                }

                string(char  *s)          //  constructor-2
                {
                        length=strlen(s);
                        name= new char[length +1];
                        strcpy(name,s);
                }

                void  display( )
                {
```

```
                cout  << name << endl;
        }

        void   join (string   &a, string  &b);
};

void  string  :  : join (string   & a, string   &b)
{
        length = a.length  + b. length;
        delete  name;

        name =  new  char[length+1];          //   dynamic allocation
        strcpy(name, a.name);
        strcpy(name,b.name);
};

int  main( )
{
        char   *first  =”AJK  ”;
        string   n1(first),n2(“COLLEGE  ”),n3(“OF ARTS AND SCIENCE”),s1,s2;
        s1.join(n1,n2);
        s2.join(s1,n3);
        n1.display( );
        n2.display( );
        n3.display( );
        s1.display ( );
        s2.display( );

        return 0;
}
```

output:

```
AJK
COLLEGE
OF ARTS AND SCIENCE
AJK COLLEGE
AJK COLLEGE OF ARTS AND SCIENCE
```

Destructors

*A destructor is a member function used to destroy the objects created by constructor.

*The destructor has the same name as the class name preceded by a tilde (~)

*It implicitly calls a delete operator.The destructor frees storage occupied by an object .

- A destructor never takes any argument nor return value.
- It will be called implicitly by the compiler when the control exits from block or function.

Example:

```
// program to explain destructor
# include <iostream.h>

int count=0;
class
{
        public:
                alpha( )
                {
                        count++;
                        cout << "Number of objects created …"<< count<<endl;
                }
                ~ alpha( )                              // …. destructor ….
                {
                        cout <<" Number of objects destroyed…"<<count<<endl;
                }
};

int main( )
{
        cout <<" Main function …."<<endl;

        alpha a1,a2,a3,a4;

        {
                cout <<" ….Block 1 …."<<endl;
                alpha a5;
        }

        {
                cout <<"….Block 2….."<<endl;
                alpha a6;
```

```
    }
    cout << "Again   to main  function…."<<endl;


    return 0;
}
```

output:

Main function…

Number of object created… 1
Number of object created… 2
Number of object created… 3
Number of object created… 4

….Block 1…

Number of object created… 5
Number of object destroyed…5

….Block 2…

Number of object created… 5
Number of object destroyed… 5

Again to main function….

Number of object destroyed …4
Number of object destroyed… 3
Number of object destroyed… 2
Number of object destroyed… 1

Note:

- The group of objects a1,a2,a3,a4 are created first and next a5 is created.
- While encountering the closing brace, the object a5 is destroyed.
- Next the object a6 is created and destroyed when the closing brace occurs.
- Similarly the rest of the objects are also destroyed.
- When the closing brace encounters, the destructors for each object in the scope are called.

Operator Overloading

- Operator overloading refers to the creation of new definition for most of the C++ operators.
- It is a mechanism to give special meanings to an operator.

- We can overload (giving additional meaning to) all the C++ operators except the following:
  - Scope resolution operator(::)
  - Sizeof operator.
  - Conditional operator(? :)
  - Class member access operator(. , .*)

Defining of operator overloading :

* In order to give additional meaning to an operator, an operator function is needed.
* General form of an operator function is :

```
return_type classname : : operator op(arguments)
{
        function statements;
}
```

- operator overloading can be carried out by means of either member function or friend function.

- A friend function will have only one argument for unary operators and two for binary operators.

- A member function has no argument for unary operators and only one or binary operators.
- This is because the object used to call a member function is passed implicity and therefore it is available for the member function.

Steps for overloading operators:

1. Create a class that defines a data type that is to be used in the overloading operation.

2. Declare the operator function operator op( ) in the public part of the class.

3. It may be either a member function or a friend function.

4.Define a operator function to do the required operation.

Example(1)

```
// Program to overload increment ++ operator
#include <iostream.h>
class counter
{
        int count;
        public:
```

```
        counter  (  )  { count=0;  }

        int display  ( ) {  return  count ;  }

        void   operator ++ ( )
        {
                count   ++;
        }

};
int   main( )
{
        counter  c1,c2;

        cout  << "C1 = "  <<   c1.display ( );
        cout  << "C2 = "  <<   c2.display ( );

        c1++;
        c2++;
        ++c2;

        cout  << " \n C1 = "  <<   c1.display ( );
        cout  << "\n C1 = "  <<   c1.display ( );

        return  0;

}
```

output:

C1=0
C2=0

C1=1
C2=2

Example (2).

```
// overloading  unary   minus  operator

class  convert
{
        int m,n;
        public:
```

```
                void receive  ( );
                void  show ( );
                void    operator   --  (  );        // …. overloaded   unary  minus…
};

void   convert  : : receive  ( )
{
        cout<<"Enter   m and n  values    :"<<endl;
        cin >>m,n;
}

void  convert  : : show ( )
{
        cout << m<<n <<endl;
}

void    convert  : : operator     -- ( )
{
        m  = - m;
        n   = - n;
}

int  main ( )
{
        convert  c;
        c. receive( );
        cout  <<"The  object   C is "<<endl;
        c.show ( );

         -- c;                    //   activate   operator     --( )

        cout  <<"\n  Now   the object  C is"  <<endl;
        c.show ( );
}
```

output :

The object C is   10 -20
Now the object C is –10   20

Overloading Binary Operators:

A friend function will have only one argument for unary operators and two for binary operators.

* A member function has no argument for unary operators and only one for binary operators.

* This is because the object used to call a member function is passed implicity and therefore it is available for the member function.

Example:

// Simple arithmetic operation of two complex numbers using overloaded binary operators.

```
Class complex
{
        float x, y;
        public:
                void   accept( )
                {
                        cout<<"Enter  two values"<<endl;
                        cin >>x>>y;
                }
                complex   operator   +  ( complex );
                complex   operator   -  ( complex );
                complex   operator   *  ( complex );
               void      display  ( );
};

// overloading   the binary operator+
complex    complex  : :  operator   +(complex  c)
{
        complex  temp;
        temp.x =  x + c.x;
        temp.y=  y  + c.y;
        return  (temp);
}

//  overloading the binary  operator  -
complex    complex  : :  operator   -(complex  c)
{
        complex  temp;
        temp.x =  x - c.x;
        temp.y=  y  - c.y;
```

```
        return  (temp);
}
//  overloading the binary operator *
complex    complex  : :  operator   *(complex  c)
{
        complex  temp;
        temp.x =  (x *c.x ) –(y*c.y);
        temp.y=  (x+ c.y) +(y*c.x);
        return  (temp);
}
void  complex::display ( )
{
        cout <<  x<<  "+i "  <<y << endl;
}

int main  ( )
{
        complex   c1,c2,c3;
        c1.accept( );
        c2.accept( );
        c3=c1+c2;                       //   calls  operator   + ( )  function
        cout <<"Addition   " <<  c3.display ( ) <<  endl;

        c3=c1-c2;                       //   calls  operator   - ( )  function
        cout <<"Subtraction  " <<  c3.display ( ) <<  endl;

        c3=c1*c2;                       //   calls  operator   * ( )  function
        cout <<"Multiplication  " <<  c3.display ( ) <<  endl;

        return  0;
}
```

Note:

Consider the following statement :

```
c3= c1+c2;
```

It invokes operator +.The object c1 takes the responsibility of calling the function (operator function) and c2 is sent as an argument. This statement is equivalent to

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

Therefore in the operator function, the members of c1are accessed directly and the data members of c2 are accessed using the dot operator.In overloading binary operators, the left- hand operator is used to call the operator Function and the right-hand operator is passed as an argument.

Overloading Binary Operators Using Friend Function

- A friend function is used to overload a binary operator with different types of operands.
- In the above example: c3=c1+c2 will work perfectly with member function.
- But c3= 10 +c2 will not work ,because the left-hand operator is responsible for calling the member function ,but it is not an object.
- In friend function ,overloaded unary operator take one argument and binary operators take two arguments.Both arguments are passed explicitly.

```
//  example for overloading * operator using friend  funcion

class assign
{
        int  x;
        public:
                void  receive( )
                {
                        cout <<"ENTER  VALUE FOR X "<<endl;
                        cin>>x;
                }
                friend   int operator  * (assign   a1,assign a2);
};

//   defining   overloaded  operator  *  using friend funcion

int operator *   (assign   a1,assign  a2)
{
        int  t=  a1.x  *  a2.x;
        return   t;
}

int  main( )
{
        assign    v1,v2;
        int res;
        cout <<"Enter   a value"<<endl;
        v1.receive ( );
        v2.receive( );
        res=v1 *  v2;  // calling   overloaded  operator   *
        cout <<  res <<endl;
        return 0;
}
```

Manipulating of strings in C++

Manipulating of strings in C++ by operator overloading using character arrays, pointers and string functions. There are no operators for manipulating the strings. There are no direct operator that could act upon the strings or manipulate the strings.

Although there are these limitations exist in C++, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal number. We can manipulate strings by operator overloading as this is not achieved by operators only.

For example :
We should be able to use statement like this in manipulating strings using operator overloading -

string3 = string1 + string2;

Let us see an simple example of above statement and use to manipulate the strings :

```
#include<iostream>

using namespace std;


int main ()

{

  string First = "This is First String and ";

  string Second = "This is Second String.";

  string Third = First + Second;

  cout << Third;

  return 0;

}
```

String Manipulation in C++ contains many string functions which we can use to manipulate the strings. In some compilers we can use them only by including iostream but in compilers which gives error have to include string library(#include<string>). Those functions can be Stringname.length , Stringname[expression](this expression is a number of character to show from the string), and many more.

program example in C++ :

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

```
int main ()
{
  string Example = "Techoschool !";

  cout << Example.length() << "\n\n";
  // This length manipulator tells the length of string entered

  cout << Example[5] << "\n\n";
  // This manipulator takes the number in parameter to show the character on that value

  cout << Example.substr(0, 5) << "\n\n";
  // This substr manipulator prints the character upto 5 character in string

  cout << Example.erase(0, 5) << "\n\n";
  // This erase the number of characters mentioned from string and print rest string

  cout << Example.insert(0, "Hello ") << "\n\n";
  // This inserts the string to exist string on the number of place of string mentioned

  cout << Example.append(5, '*') << "\n\n";
  // This appends the given string to the existing string from the last position

  cout << Example.replace(0, 5, "Replaced Character");
  // This replace string according to position entered with the string mentioned

  return 0;
}
```

RULES FOR OVERLOADING OPERATOR :-

1) Operators already predefined in the C++ compiler can be only overloaded.
2) Overloading cannot change operator's templates.for eg: the increment operator ++ is used only as unary operator, it cannot be used as binary operator.
3) Overloading an operator does not change its basic meaning. for eg: the ++ operator cannot be overloaded to subtract two objects.
4) Unary operator, overloaded by means of a member of a member function, takes no explicit argument and return no explicit values.But those overloaded by means of a friend function take one reference argument (the object of the relevent class).
5) Binary operators overloaded through a member function take one explicit argument and those overloaded through a friend function take two explicit arguments.

Type Conversion in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion** Also known as ‘automatic type conversion’.
    - Done by the compiler on its own, without any external trigger from the user.
    - Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
    - All the data types of the variables are upgraded to the data type of the variable with largest data type.
    - bool -> char -> short int -> int ->
    -
    - unsigned int -> long -> unsigned ->
    -
    - long long -> float -> double -> long double
    - It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:

```
// An example of implicit conversion
#include <iostream>
using namespace std;
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    // x is implicitly converted to float
    float z = x + 1.0;

    cout << "x = " << x << endl
        << "y = " << y << endl
        << "z = " << z << endl;

    return 0;
}
```

Output:

```
x = 107
y = a
z = 108
```

2. **Explicit Type Conversion**: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

- **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Syntax:

(type) expression

where *type* indicates the data type to which the final result is converted.

Example:

```
// C++ program to demonstrate
// explicit type casting

#include <iostream>
using namespace std;

int main()
{
   double x = 1.2;

   // Explicit conversion from double to int
   int sum = (int)x + 1;

   cout << "Sum = " << sum;

   return 0;
}
```

Output:

Sum = 2

- **Conversion using Cast operator:** A Cast operator is an **unary operator** which forces one data type to be converted into another data type. C++ supports four types of casting:
  1. Static Cast
  2. Dynamic Cast
  3. Const Cast
  4. Reinterpret Cast

Example:

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

```
int main()
{
   float f = 3.5;

   // using cast operator
   int b = static_cast<int>(f);

   cout << b;
}
```

Output:

3

Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.

REFERENCE:

1. E. Balaguruswamy , " Object oriented programming with C++", TataMcGraw Hill publishing company Limited, 1998.
2. K.R, Venugopal, Rajkumar, T. Ravishankar, "Mastering C++", tata mc graw – Hill publishing company Limited, 1998.
3. D. Ravichandran, Programming with C++", Tata McGraw – Hill published Company Limited.

Prepared By Dr.N.Shanmugavadivu