09
10
19

UNIT - 3

OPERATOR OVERLOADING

* C++ tries to make the user defined datatypes to behave in the same way as the build (pre) or built in datatypes

* OO provides a flexible option for the creation of the new definition for most of the C++ operators. Some of the operators that cannot be overloaded in C++ are as following:

i) scope Resolution operator (::)
ii) ternary operators CONDITIONALS
iii) sizeof operator (sizeof)
iv) Class member access operator (.*).

* RULES of OPERATOR OVERLOADING:

i) We cannot use an operator in a manner that changes the syntax for the original operator.

ii) We cannot create new operator symbols.

* DEFINING OPERATOR OVERLOADING:

```
returntype classname :: operator op (argument list)
                        KEYWORD   ↓
                                OPERATOR SYMBOL
   { - -
      - - }
```

** Overloading UNARY OPERATORS:

```
class Space
{ int x;
    int y;
    public: void input (int a, int b)
              { x = a;
                y = b;
              }
```

```
        void display ()     or void output ()
        { cout << x << y ;
                                      → OPERATOR SYMBOL
        }
        void operator - () ;          → DECLARATION
    };
    void Space :: operator - ()    }  DEFINITION
    {    x = -x ;                  }    OF
         y = -y ;                  }  - OPERATOR
    }                              }  OVERLOADING
    void main ()
    { Space s1 ;
      s1. input (10, -20) ;
      s1. output () ;
       - s1 ;         → CALLING OF OPERATOR FUNCTION
      s1. output () ;
      getch () ;
    }
```

* overloading UNARY OPERATOR using FRIEND FUNCTION.

```
    class space
    {  int x ;
       int y ;
       public : void input (int a, int b)
                {  x = a ;
                   y = b ;
                }
                void output ()
                {  cout << x << y ;
                }
DECLARATION OF
FRIEND FUNCTION ←   friend void operator - (space &s) ;
    } ;
```

COPIED
OBJECT
↑

```
void operator - (space &s)
{ s.x = -s.x;
  s.y = -s.y;
}
```

NOTE : The function **Operator -()** takes no arguments still it changes the sign of the data members of the object s1. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

↳ FOR 1st PROGRAM.

** **Overloading BINARY OPERATORS :**

```
    class complex
    { float x;
      float y;
      public: complex (float real, float imag)
                 { x = real;
   Constructor     y = imag;

complex (class)  }
used as a return
     type.      complex operator + (complex &c);
display function > void display ()
                   { cout << x << y ;}
                                        function name
   }; return    class
       type      name
   complex complex :: operator + (complex &c)
   { complex temp;  → temp object of       arguments
                      complex class.          passed.
      temp.x = x + c.x;
      temp.y = y + c.y;
      return temp;

   }
```

```
void main ()
{ complex c1 , c2 , c3 ;
    c1 = complex (2.5 , 3.5) ;      & c1.complex (2.5 , 3.5) ;
    c2 = complex (7.5 , 12.0) ;
    c3 = c1 + c2 ;                   c3 = c1.operator + (c2) ;
    c1. display () ;               → working like this
    c2. display () ;
    c3. display () ;    + function
    getch () ;                       argument for +
}                                              function
values in + function (x and y).
```

* Overloading BINARY OPERATOR using FRIEND FUNCTION:

```
    class complex
    { float x ;
      float y;
      public : complex (float real, float imag)
              { x = real;
                 y = imag ;
              }
      friend complex operator + (Complex &c1 , complex &c2)
       void display ()
        { cout << x << y ; }
    };
    complex operator + (complex &c1 , complex &c2)
    {   complex temp ;
        temp. x = c1. x + c2. x ;
        temp. y = c2. y + c1. y ;
         return temp ;
    }
```

16.10
2019

TYPE · CONVERSION

PAGE:
DATE: / /

* 3 types of Conversions are there :

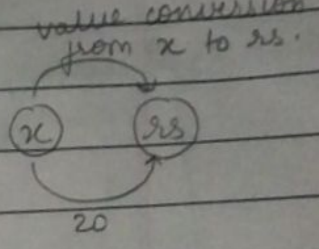
1) Conversion from Basic datatype to Class type.

11) Conversion from Class type to Basic datatype.

111) Conversion from Class type to Class Type.

* CONVERSION FROM BASIC to CLASS TYPE :

```
class Money
{ int rs;
   public : Money (int x)
              { rs = x ; }
                  ↓
};            Conversion
  void main ()
  { Money M1;
     M1 (20);
     getch ();
  }
```



* CONVERSION FROM CLASS TYPE to BASIC TYPE :

C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type.

SYNTAX for creating casting operator :

```
        operator  type-name ()
          {            └ };  →  in which you want to
                                    convert.
```

* PROGRAM :

```
  class student
  { int roll_no;
    public :   student (int x)
               { roll_no = x+10 ; }
               operator float ()        // CASTING OPERATOR
                                          DEFINITION.
```

```
{ return float (roll_no)/2 ;
}
void show ()
{ cout << roll_no ; }
};
void main ()
                    → float declaration
{ float f;
  student s1 (20);              → Parameterised Constructor.
  s1. show ();
  f = s1;                       ⇒ OBJECT → BASIC.
  cout << f;   getch ();                  ↓
                                  // casting or
}                                 type conversion.
```

value of float () returned to f.

* CONVERSION FROM CLASS TYPE to CLASS TYPE :

creating casting operator function.

```
class minute              → SOURCE CLASS
{ int m;
  public : minute ( int x)          → to be converted in
                                      which class.
           { m = x ; }
           operator hour ()  ]
           { hour h1 ;       ]   CASTING
             h1.h = m/60 ;   ]   FUNCTION
             return (h1) ;   ]
           }
  void show ()
  { cout << " Minutes = " << m; }
  };
class hour                    → DESTINATION CLASS.
{ int h ;
```

```
public : hour()
           { h = 0;        ]  default constructor
           }
           void show()
             { cout <<h;  }
};
void main()
{  minute min (60);
   hour hr;
   hr = min;                      → TRANSFERRING or CONVERTING
                                            VALUES.
60//  min.show();
1//   hr.show();   getch();
}
```

UNIT 3 finished

17.10
2019

INHERITANCE

UNIT-4

PAGE:
DATE: / /

* single level Inheritance :

```
class A
{          };
class B : public A        → INHERITING CLASS A to class B
{          };
void main ()
{       getch (); }
```

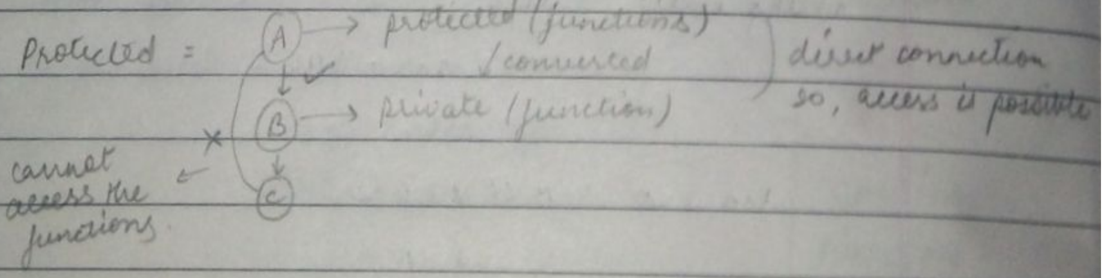
VISIBILITY MODE

- Three visibility modes:

1) Public   11) Private   111) Protected.

Visibility Mode → Class Member ↓	PRIVATE	PUBLIC	PROTECTED
* PRIVATE	Never Inherited	never Inherited	never Inherited
* PROTECTED	Private	Protected	Protected
* PUBLIC	Private	Public	Protected

- Protected =


A
protected (functions) / connected
B
private (function)
C
direct connection so, access is possible
cannot access the functions

- class Members = Both functions

Ex :

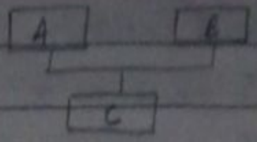
```
class one                          -BASE CLASS.
{ int x ;
  public : void readx ()
             { cin >> x ;  }
void showx
  { cout << x ;
  }
};
 class two : public one                    - DERIVED CLASS
  { int y ;
     public : void ready ()
                { readx (); cin>>y;
                }
                void showxy ()
                { showx ();
                    cout << y;  }
  };
  void main ()
  { two t ;
     t. readxy ();
     t. showxy ();   getch ();
  }
```

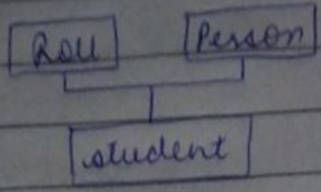
→
```
class one
 { protected : int x;
     public : same as above ;
 };
  class two : public one
   { protected : int y;
       public : void reaxy ()
                  { cin >> x ;   cin >>y ; }
                void showxy ()
                  { cout <<x  ; cout <<y ; }
  };
```

x(one)
↓ protected
x(two)

PAGE :
DATE : / /

* Multiple Inheritance :





```
class roll
{ protected : int r ;
  public : void read ()
             { cin >> r ; }
             void show ()                          ] NO
           { ~~cin >> r~~ ; }  cout << name;       ] need
};
class Person
{ protected : char name [20] ;
  public : void readn ()
             { cin >> name; }                      ] NO
             void shown ()                         ] NEED
             { cout << name ; }
};      into this              from these  → INHERITANCE
class ~~void~~ student : public roll, public person
{ protected : int marks ;
  public : void reads ()
             { cin >> r >> name >> marks ; }
             void shows ()
           { cout << r << name << marks ; }
};
void main ()
{ student s1 ;
  cout << "Enter details : " ;
    s1. reads () ;    s1. shows () ;    getch () ; }
```