Object Oriented Programming With C++

MODULE-3: Polymorphism

Compiled by: Dr. Amita Jain

amita.jain@vsit.edu.in

VSIT | Vidyalankar School of Information Technology

NAAC Accredited College

Vidyalankar School of Information Technology
Wadala (E), Mumbai

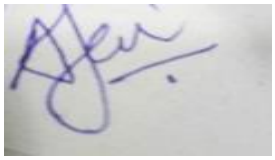
www.vsit.edu.in

VSIT | Vidyalankar School of Information Technology

NAAC Accredited College

Certificate

This is to certify that the e-book titled "OBJECT ORIENTED PROGRAMMING WITH c++" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.



Signature

Date: 18-11-2019

Dr. Amita Jain

Assistant Professor

Department of IT

⚠ *DISCLAIMER: The information contained in this e-book is compiled and distributed for educational purposes only. This e-book has been designed to help learners understand relevant concepts with a more dynamic interface. The compiler of this e-book and Vidyalankar Institute of Technology give full and due credit to the authors of the contents, developers and all websites from wherever information has been sourced. We acknowledge our gratitude towards the websites YouTube, Wikipedia, and Google search engine. No commercial benefits are being drawn from this project.*

Unit III: Polymorphism

Contents:

- Polymorphism: Concept of function overloading,
- overloaded operators
- overloading unary operators
- overloading binary operators,
- overloading comparison operator,
- overloading arithmetic assignment operator,
- Data Conversion between objects and basic types
- Virtual Functions: Introduction and need,
- Pure Virtual Functions,
- Static Functions,
- this Pointer,
- abstract classes,
- virtual destructors.

Recommended Books

Texts Books:

- Schaum's Outline Of Theory And Problems Of Programming With C++ .
  By John R.Hubbard, Tatamcgraw-Hill
- Object Oriented Programming with C++,
  By E.Balagurusamy,Fourth Edition, TATAMCGRAW-HILL.
- Object Oriented Programming with C++,
  By P. Sarang 2nd Edition,(PHI) EEE edition

Reference Books:

- C++ programming , 3rd Edition, Bjarne Stroustrup
- Mastering C++, 2nd Edition, Venugopalan, TataMcgrawHill
- C++ Programming, , Robert Lafore,

- C++ for Beginners, P. M. Harwani, X-Team Series,

Prerequisites and Linking

	Pre-requisites	Linking			
Unit I	Sem. I	Sem. III	Sem. IV	Sem. V	Sem. VI
Object Oriented Methodology and Principle of OOPS	Imperative Programming	Python and Data Structures	Core Java	Advanced Web Programming, Enterprise Java & Project	Project

Notes with video links:

Polymorphism

The word polymorphism means having many forms. C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Function Overloading

If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

Ways to overload a function

1. By changing number of Arguments.
2. By having different types of argument.

Number of Arguments different

In this type of function overloading we define two functions with same names but different number of parameters of the same type. For example, in the below mentioned program we have made two sum() functions to return sum of two and three integers.

```
int sum (int x, int y)
{
 cout << x+y;
}

int sum(int x, int y, int z)
{
 cout << x+y+z;
}
```

Here sum() function is overloaded, to have two and three arguments. Which sum() function will be called, depends on the number of arguments.

```
int main()
{
sum (10,20);  // sum() with 2 parameter will be called

sum(10,20,30);  //sum() with 3 parameter will be called
}
```

Different Datatype of Arguments

In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different. For example in this program, we have two sum() function, first one gets two integer arguments and second one gets two double arguments.

```
int sum(int x,int y)
{
 cout<< x+y;
}

double sum(double x,double y)
{
 cout << x+y;
}
```

```
int main()
{
 sum (10,20);
 sum(10.5,20.5);
}
```

Operator Overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function, which describes the task. The general form of an operator function is:

```
return type classname : : operator op(arglist)
{
        Function body                          // task defined
}
```

where return type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator. operator op is the function name.

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototypes as follows :

```
vector operator+(vector);                  //vector addition
vector operator-();                            //unary minus
friend vector operator+(vector,vector);    //vector addition
friend vector operator-(vector);               //unary minus
vector operator-(vector &a);                   //subtraction
int operator==(vector);                        //comparison
```

friend int operator==(vector,vector) //comparison

vector is a data type of class and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics).

The process of overloading involves the following steps:

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

Overloaded operator functions can be invoked by expressions such as

op x or x op l

for unary operators and

x op y

for binary operators. op x (or x op) would be interpreted as

operator op (x)

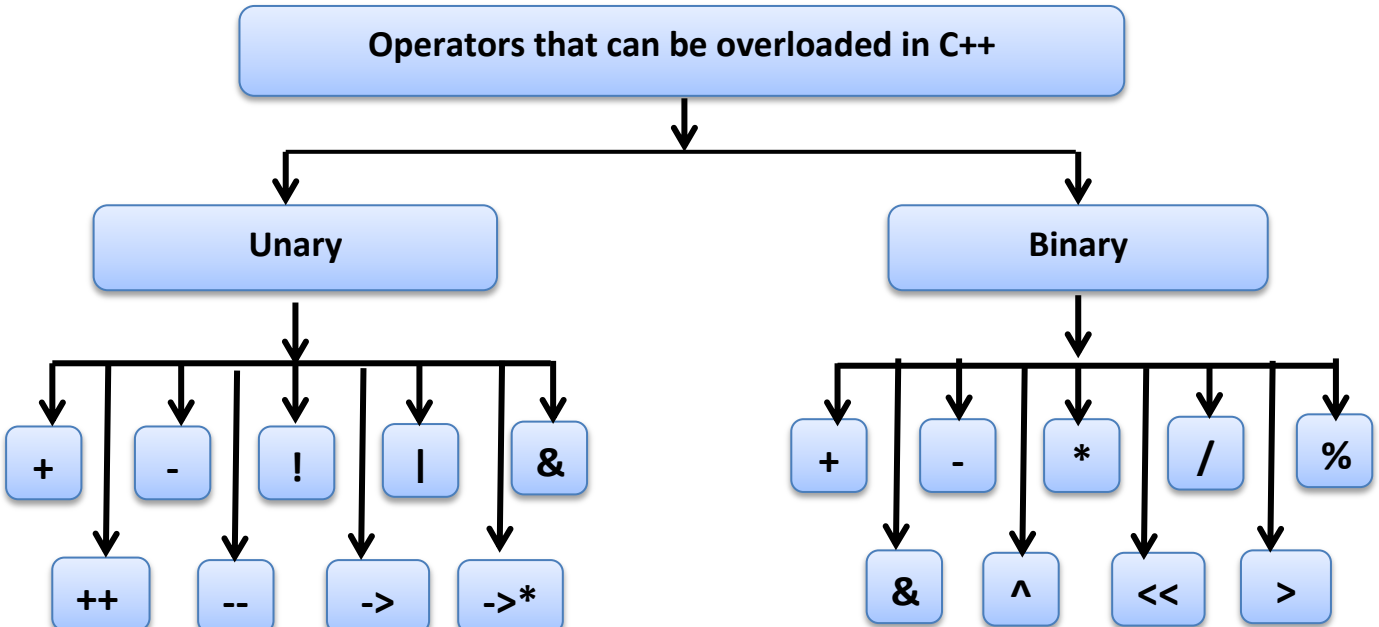
for friend functions. Similarly, the expression x op y would be interpreted as either

x.operator op (y)

in case of member functions, or

operator op(x, y)

in case of friend functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.



Overloadable/Non-overloadableOperators:

Following is the list of operators which can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded:

::	.*	.	?:

OVERLOADING UNARY OPERATORS

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 it 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.

Program : shows how the unary minus operator is overloaded.

```
#include <iostream>

using namespace std;

class space

{

    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 « "\n";
}
void space :: operator-( )
{
    x = -x;
    y = -y;
    z = -z;
}
int main()
{
    space S;
```

```
    S.getdata(10, -20, 30);
    cout « "S : ";
    S.displayO;
    -S;                                    // activates operator-() function
    cout << "S : ";
    S.displayO;
    return 0;
}
```

The Program produces the following 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.

Remember, a statement like

```
S2 = -S1;
```

will not work because, the function operator-() does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend    void operator-(space &s);          // declaration
void operator-(space &s)                          // definition
{
        s.x = -s.x;
        s.y = -s.y;
        s.z = -s.z;
}
```

Note : Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed

to operator-*). Therefore, the changes made inside the operator function will not reflect in the called object.

Video: Unary Operator Overloading

Link: https://www.youtube.com/watch?v=zBmdsB8C6AA

OVERLOADING BINARY OPERATORS

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. In Chapter 6, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum(A, B);                    // functional notation
```

was used. The functional notation can be replaced by a natural looking expression

```
C = A + B;                        // arithmetic notation
```

by overloading the + operator using an operator+() function. The below Program illustrates how this is accomplished.

Operloading + Operator

```
#include <iostream>
```

```
using namespace std;
class complex
{
    float x;                                    //real part
    float y;                                    //imaginary part
    public:
    complex(){ }                                //constructor 1
    complex(float real, float imag)     //constructor 2
    { x = real; y = imag; }
};
complex operator+(complex c);
{
    complex temp;                               //temporary
    temp.x = x + c.x;                           //these are
    temp.y = y + c.y;                           //float additions
    return(temp);
}
void complex :: display(void);
{
            cout << x << " + j" << y << "\n";
}
int main( )
{
    complex C1, C2, C3;                 //invokes constructor 1
    C1 = complex (2.5, 3.5);                    //invokes constructor 2
    C2 = complex (1.6, 2.7);
    C3 = C1 + C2;
    cout << "C1 ="; C1.display( );
```

```
    cout << "C2 ="; C2.display( );
    cout << "C3 ="; C3.display( );
    return 0;
}
```

The output of Program would be

```
C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.2
```

Note : Let us have a close look at the function operator+() and see how the operator overloading is implemented.

```
complex complex :: operator+(complex c)
{
    complex temp;
    temp.x = x + c.x;
    temp.y = y + c.y;
    return(temp);
}
```

We should note the following features of this function:

1) It receives only one complex type argument explicitly.

2) It returns a complex type value.

3) It is a member function of complex.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
C3 = CI + C2;                    //invokes operator+() function
```

We know that a member function can be invoked only by an object of the same class. Here, the object CI takes the responsibility of invoking the function and C2 plays the role of an argument that is passed to the function. The above invocation statement is equivalent to

```
C3 = Cl.operator+(C2);                    // usual function call syntax
```

Therefore, in the operator+() function, the data members of CI are accessed directly and the data members of C2 (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
temp.x = x + c.x;
```

c.x refers to the object C2 and x refers to the object CI. temp.x is the real part of temp that has been created specially to hold the results of addition of CI and C2. The function returns the complex temp to be assigned to C3. Figure shows how this is implemented.

As a rule, in overloading of binary operators, the left-hand operand is used to invoke the operator function and the right- hand operand is passed as an argument.

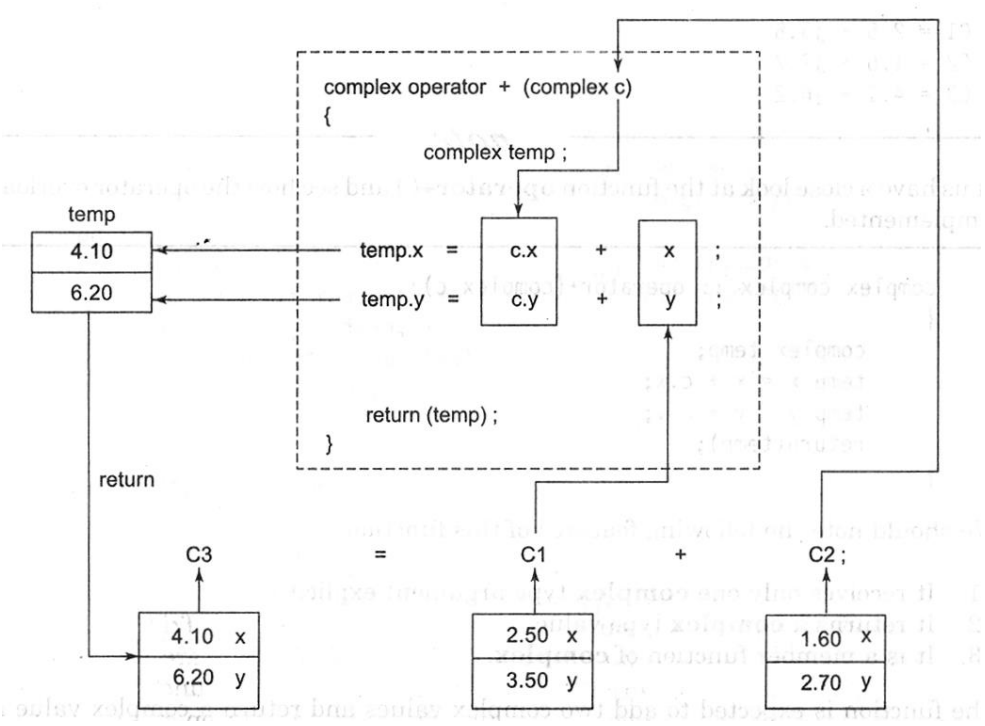


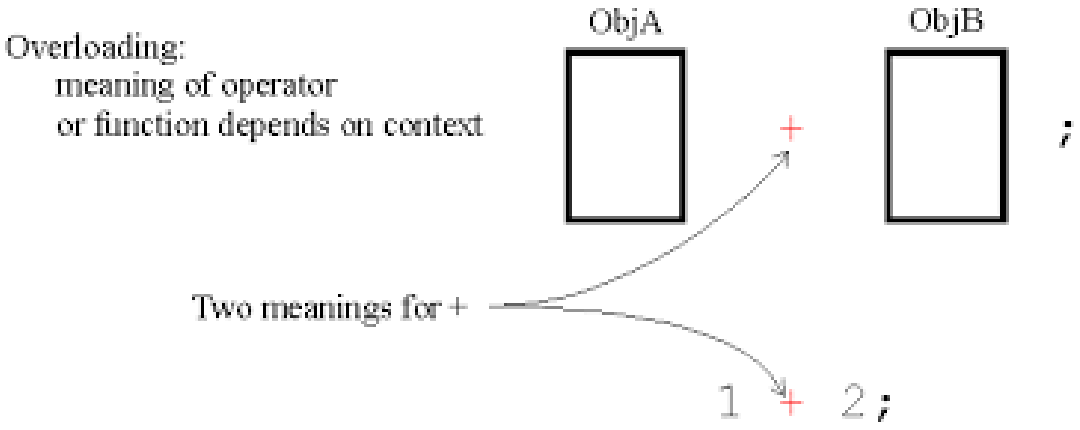
Fig. : Implementation of the overloaded + operator.

We can avoid the creation of the temp object by replacing the entire function body by the i..I lowing statement:

```
return complex((x+c.x),(y+c.y));      // invokes constructor 2
```

What does it mean when we use a class name with an argument list? When the compiler pomes across a statement like this, it invokes an appropriate constructor, initializes an Object with no name and returns the contents for copying into an

object. Such an object is in I led a temporary object and goes out of space as soon as the contents are assigned to another object. Using temporary objects can make the code shorter, more efficient and better to read.



Overloading the Assignment Operator

Of all the operators, the assignment operator = is probably used the most. Its purpose is to copy one object to another. Like the default constructor, the copy constructor, and the destructor, the assignment operator is created automatically for every class that is defined. But also like those other three member functions, it can be defined explicitly in the class definition.

Example : Adding an Assignment Operator to the Ratio Class

Here is a class interface for the Ratio class, showing the default constructor, the copy constructor, and the assignment operator :

```
class Ratio
{   public:
        Ratio(int = 0,  int = 1);           // default constructor
        Ratio(const Ratio&);                // copy constructor
        void operator=(const Ratio&);       // assignment operator
        // other declarations go here
    private:
        int num,  den;
};
```

Note the required syntax for the assignment operator. The name of this member function is operator=. Its argument list is the same as that of the copy constructor: it contains a single argument of the same class, passed by constant reference.

Here is the implementation of the overloaded assignment operator:

```
void Ratio :: operator = (const Ratio& r)
{   num = r.num;
    den = r.den;
}
```

It simply copies the member data from the object r to the object that owns the call.

6) What is "this" pointer?

THE 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() will set the pointer this to the address of the object A. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called The pointer this acts as an implicit argument to all the member functions. Consider the following simple example :

```
Class ABC
{
    int a;
    ……
    ……
};
```

C++ allows assignments to be chained together, like this:

```
x = y = z = 3.14;
```

This is executed first by assigning 3.14 to z, then to y, and finally to x. But, as Example 1 shows, the assignment operator is really a function named operators In this chain,

the function is called three times. On its first call, it assigns 3.14 to z, so the input to the function is 3.14. On its second call, it assigns 3.14 to y, so its input again must be 3.14. So that value should be the output (i.e., return value) of the first call. Similarly, the output of the second call should again be 3.14 to serve as the input to the third call. The three calls to this function are nested, like this:

```
f(x, f(y, f(z, 3.14)))
```

The point is that the assignment operator is a function that should return the value it assigns. Therefore, instead of the return type void, the assignment operator should return a reference to the same type as the object being assigned:

```
Ratio& operator = (Ratio& r)
```

OVERLOADING ARITHMETIC OPERATORS

All programming languages provide the standard arithmetic operators +,–, *, and / for numeric types. So it is only natural to define these for user-defined numeric types like the Ratio class. In older programming languages like C and Pascal, this is done by defining functions like this:

```
Ratio product (Ratio x,  Ratio y)
{    Ratio z(x.num*y.num,  x.den*y.den);
     return z;
}
```

This works. But the function has to be called in the conventional way:

```
z = product(x, y);
```

C++ allows such functions to be defined using the standard arithmetic operator symbols, so that they can be called more naturally:

```
z = x*y;
```

Like most operators in C++, the multiplication operator has a function name that uses the reserved word operator: its name is "operator*". Using this in place of "product" in the code above, we would expect the overloaded function to look something like this:

```
Ratio operator* (Ratio x, Ratio y)
{    Ratio z (x.num*y.num, x.den*y.den);
     return z;
```

```
}
```

But this is not a member function. If it were, we would have to set it up as in with only one argument. The operator* function requires two arguments.

Since the overloaded arithmetic operators cannot be member functions, they cannot access the private member data num and den. Fortunately, C++ allows an exception to this rule so that we can complete our definitions of the overloaded arithmetic functions. The solution is to declare the function as a friend of the Ratio class.

A friend function is a nonmember function that is given access to all members of the class within which it is declared. So it has all the privileges of a member function without actually being a member of the class. This attribute is used mostly with overloaded operators.

Example : Declaring the Multiplication Operator as a friend Function

Here is the Ratio class declaration with the overloaded multiplication operator declared as a friend function:

```
class Ratio
{           friend Ratio operator* (const Ratio&, const Ratio&);
    public:
        Ratio(int = 0,  int = 1);
        Ratio(const Ratio&);
        Ratio& operator = (const Ratio&);
        // other declarations go here
    private:
        int num,  den;
        // other declarations go here
};
```

Note that the function prototype is inserted inside the class declaration, above the public section. Also note that the two arguments to the function are both passed by constant reference.

Now we can implement this nonmember just as we had expected:

```
Ratio operator*(const Ratio& x, const Ratio& y)
```

```
{   Ratio z(x.num * y.num, x.den * y.den);
    return z;
}
```

Note that the keyword friend is not used in the function implementation. Also note that the scope resolution prefix Ratio: : is not used because this is not a member function.

OVERLOADING THE ARITHMETIC ASSIGNMENT OPERATORS

C++ allows you to combine arithmetic operations with the assignment operator; for example, using x *= y in place of x = x * y. These combination operators can all be overloaded for use in your own classes.

Example : The Ratio Class with an Overloaded *= Operator

```
class Ratio
{   public:
        Ratio(int = 0,  int *1);
        Ratios operator = (const Ratios);
        Ratios operator*= (const Ratios);
        // other declarations go here
    private:
        int num, den;
        // other declarations go here
};
Ratio& Ratio : : operator* = (const Ratios r)
{   num = num*r.num;
    den = den*r.den;
    return *this;
 }
```

The operator operator* = has the same syntax and nearly the same implementation as the basic assignment operator operators By returning * this, the operator can be chained, like this:

```
x * = y * = z ;
```

It is also important to ensure that overloaded operators perform consistently with each other. For example, the following two lines should have the same effect, even though they call different operators:

```
x = x * y;
x *= y
```

Overloading the Relational Operators

The six relational operators <, >, <=, >=, = =, and > = can be overloaded the same way that the arithmetic operators are overloaded: as friend functions.

Example : Overloading the Equality Operator = = in the Ratio Class

Like other friend functions, the equality operator is declared above the public section of the class:

```
class Ratio
{           friend int operator = =(const Ratios,  const Ratios);
        friend Ratio operator*(const Ratios,  const Ratios);
        // other declarations go here
    public :
        Ratio(int = 0,  int = 1);
        Ratio(const Ratios);
        Ratios operator = (const Ratios);
        // other declarations go here
    private:
        int num, den;
        // other declarations go here
};
int operator = =(const Ratios x, const Ratios y)
{   return (x.num * y.den = = y.num * x.den);
}
```

The test for equality of two fractions a/b and c/d is equivalent to the test a*d == b*c. So we end up using the equality operator for ints to define the equality operator for Ratios.

Note that the relational operators return an int type, representing either "true" (1) or "false" (0).

Video: Relational Operator Overloading

Link : https://www.youtube.com/watch?v=cyOrNgkdc2E

OVERLOADING THE STREAM OPERATORS

C++ allows you to overload the stream insertion operator >> for customizing input and the stream deletion operator << for customizing output. Like the arithmetic and relational operators, these should also be declared as friend functions.

For a class T with data member d, the syntax for the output operator is

```
friend ostream& operator <<(ostream& ostr, const T& t)
{    return ostr << t.d;    }
```

Here, ostream is a standard class defined (indirectly) in the iostream.h header file. Note that all the parameters and the return value are passed by reference.

This function can then be called using the same syntax that we used for fundamental types:

```
cout << "x= " <<x<< ", y = " << y << endl;
```

Here is an example of how custom output can be written:

Example : Overloading the Output Operator << the Ratio Class

```
class Ratio
{           friend ostream& operator <<(ostream&,  const Ratio&);
    public:
        Ratio(int n = 0, int d = l) : num(n), den(d) {  }
        // other declarations go here
    private:
        int num, den;
        // other declarations go here
};
int main( )
{   Ratio x(22,7),  y(–3,8);
    cout << "x = " << x << ", y = " << y << endl;
}
    ostream& operator <<(ostreamS ostr,  const Ratios r)
{   return ostr << r.num << '/'  << r.den;
}
```

Output :

x = 22/7, y = –3/8

Type conversion:

In our original implementation of the Ratio class, we defined the member function convert ( ) to convert from type Ratio to type double:

```
double convert( ) {       return double(num)/den;    }
```

This requires the member function to be called as

```
x.convert ( );
```

In keeping with our goal to make objects of the Ratio class behave like objects of fundamental types (i.e., like ordinary variables), we would like to have a conversion function that could be called with a syntax that conforms to ordinary type conversions:

```
n = int(t);
y = double(x);
```

This can be done with a conversion operator.

Our Ratio class already has the facility to convert an object from int to Ratio:

```
Ratio x(22);
```

This is handled by the default constructor, which assigns 22 to x. num and 1 to x. den. This constructor also handles direct type conversions from type int to type Ratio:

```
x = Ratio(22);
```

Constructors of a given class are used to convert from another type to that class type.

To convert from the given class type to another type requires a different kind of member function. It is called a conversion operator, and it has a different syntax. If type is the type to which the object is to be converted, then the conversion operator is declared as

```
operator type( );
```

For example, a member function of the Ratio class that returns an equivalent float would be declared as

```
operator float( );
```

Or, if we want it to convert to type double, then we would declare it as

```
operator double( );
```

And, if we want it to be usable for constant Ratios (like pi), then we would declare it as

```
operator double ( ) const;
```

In our implementation of the Ratio class we defined the member function convert () for this purpose.

Example : Adding a Conversion Operator to the Ratio Class

```
class Ratio
{   friend istream& operator >>(istream&, Ratios);
    friend ostream& operator <<(ostream&, const Ratios);
    public:
        Ratio(int n=0,  int d = l) : num(n), den(d)   {  }
        operator double( ) const;
    private :
            int num, den;
};

int main ( )
{   Ratio x (–5, 8)
    cout << "x = " << x << ", double (x) = " << double (x) << endl;
    const Ratio P(22,7);
    const double PI = double(P);
    cout << "P = " << P << ",  PI = " << PI << endl;
}
Ratio :: operator double( ) const
{   return double(num)/den;
}
x = –5/8, double(x) = –0.625
p = 22/7, pi = 3.14286
```

First we use the conversion operator double ( ) to convert the Ratio object x into the double –0.625. Then we use it again to convert the constant Ratio object p into the constant double pi.

Overloading the Increment and Decrement Operators

The increment operator ++ and the decrement operator – – each have two forms: prefix and postfix. Each of these four forms can be overloaded. We'll examine the overloading of the increment operator here. Overloading the decrement operator works the same way.

When applied to integer types, the pre-increment operator simply adds 1 to the value of the object being incremented. This is a unary operator: its single argument is the object being incremented. The syntax for overloading it for a class named `T` is simply

```
T operator++( );
```

So for our Ratio class, it is declared as

```
Ratio operator++( );
```

Example : Adding a Pre-increment Operator to the Ratio Class

This example adds an overloaded pre-increment operator ++ to our Ratio class. Although we can make this function do whatever we want, it should be consistent with the action that the standard pre-increment operator performs on integer types. That adds 1 to the current value of the object before that value is used in the expression. This is equivalent to adding its denominator to its numerator:

$$\frac{22}{7}+1=\frac{22+7}{7}=\frac{29}{7}$$

So, we simply add den to num and then return * this, which is the object itself:

```
class Ratio
{           friend ostream& operator <<(ostream&,  const Ratio&);
    public:
        Ratio(int n = 0,  int d = l) : num(n),  den(d) {  }
        Ratio operator++( );
        // other declarations go here
    private:
        int num,  den;
        // other declarations go here
```

```
};
int main( )
{   Ratio x(22, 7),  y = ++x;
    cout << "y = " << y << ", x = " << x << endl;
}
Ratio Ratio : : operator++( )
{   num += den;
    return *this;
}
```

Output :

y = 29/7, x = 29/7

Example : Adding a Post-Increment Operator to the Ratio Class

To be consistent with the ordinary post-increment operator for integer types, this overloaded version should not change the value of x until after it has been assigned to y. To do that, we need a temporary object to hold the contents of the object that owns the call. This is done by assigning *this to temp. Then this object can be returned after adding den to num.

```
class Ratio
{   friend ostream& operator<<(ostream&,  const Ratio&);
    public:
        Ratio (int n = 0, int d = 1) : num(n), den(d) { }
        Ratio operator++( );              // pre- increment
        Ratio operator++(int);            // post-increment
    private:
        int num, den;
};
int main( )
{   Ratio x(22, 7),  y = x++;
    cout << "y = " << y << ", x = " << x << endl;
```

```
}
Ratio Ratio :: operator++(int)
{   Ratio temp = *this;
    num += den;
    return temp;
}
```

y = 22/7, x = 29/7

Overloading the Subscript Operator

Recall that, if a is an array, then the expression a[i] really means nothing more than *(a+i). This is because a is actually the address of the initial element in the array, so a + i is the address of the ith element, since the number of bytes added to a is i times the size of each array element.

The symbol [ ] denotes the subscript operator. Its name derives from the original use of arrays, where a[i] represented the mathematical symbol a_i. When used as a [i], it has two operands: a and i. The expression a[i] is equivalent to operator [ ] (a, i). And as an operator, [ ] can be overloaded.

Example : Adding a Subscript Operator to the Ratio Class

```
#include <iostream>
#include <stdlib>                    // defines the exit( ) function
class Ratio
{           friend ostream& operator <<(ostream&, const Ratio&);
    public:
    Ratio(int n = 0,  int d = 1) : num(n),  den(d) {  }
    int& operator [ ] (int);
    // other declarations go here
    private:
    int num, den;
    // other declarations go here
} ;
```

```
int main( )
{    Ratio x(22, 7);
     cout << "x = " << x << endl;
     cout <<  "x[l]  = " << x[l]  <<  ",  x[2]   = " << x[2]  << endl;
}
int& Ratio :: operator[ ](int i)
{    if  (i = = 1)  return num;
     if  (i = = 2)  return den;
     cerr << "ERROR: index out of range\n";
     exit(0);
}
```

Output :

x = 22/7

x[1] = 22, x[2] = 7

Video: Operator Overloading

Link : https://www.youtube.com/watch?v=XQKamEpjk4E

Virtual Function

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

```
class Shape {
  protected:
    int width, height;
  public:
    Shape( int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    // pure virtual function
    virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

VIRTUAL FUNCTIONS AND POLYMORPHISM

One of the most powerful features of C++ is that it allows objects of different types to respond differently to the same function call. This is called polymorphism and it is achieved by means of virtual functions. Polymorphism is rendered possible by the fact that a pointer to a base class instance may also point to any subclass instance:

```
class X
{    // ...
}
class Y : public X                          // Y is a subclass if X
{    // ...
}
int main( )
{    x* p;                                  // P is a pointer to objects of
base class X
     Y y;
     p = &y;                                // p can also point to objects
of subclass Y
}
```

So if p has type x* ("pointer to type x"), then p can also point to any object whose type is a subclass of x. However, even when p is pointing to an instance of a subclass Y, its type is still x*. So an expression like p – >f ( ) would invoke the function f ( ) defined in the base class.

Recall that p –> f( ) is an alternate notation for (*p). f ( ). This invokes the member function f( ) of the object to which p points. But what if p la actually pointing to an object y of a subclass of the class to which p points, and what if that subclass Y has its own overriding version of f ( ) ? Which f( ) gets executed: X: : f ( ) or Y: : f ( )? The

answer is that p –>f ( ) will execute X:: t ( ) because p had type X*. The fact that p happens to be pointing at that moment to an instance of subclass Y is irrelevant; it's the statically defined type X* of p that normally determines its behavior.

This demo program declares p to be a pointer to objects of the base class X. First it assigns p to point to an instance x of class X. Then it assigns p to point to an instance y of the derived class Y.

```
class X
{    public:
          void f( ) { cout << "X: :f( ) executing\n"; }
};

class Y : public X
{    public:
          void f( ) { cout << "Y::f( ) executing\n"; }
};

int main( )
{    X x;
     Y y;
     X* p = &x;
     p –> f ( );              // invokes X: :f( ) because p has type X*
     p = &y
     p –> f ( )               // invokes X: :f( ) because p has type X*
}
X: :f( ) executing
X: :f ( ) executing
```

Two function calls p – >f ( ) are made. Both calls invoke the same version of f ( ) that is defined in the base class X because p is declared to be a pointer to X objects. Having p point to y has no effect on the second call p –>f ( ).

Transform X: : f () into a virtual function by adding the keyword "virtual" to its declaration:

```
class X
{       public:
        virtual void f( ) { cout << "X: :f( ) executing\n"; }
};
```

With the rest of the code left unchanged, the output now becomes

X: :f( ) executing

Y: :f( ) executing

Now the second call p –>f( ) invokes Y: :f( ) instead of X: :f( ).

1. What is mean by virtual destructors?

VIRTUAL DESTRUCTORS

Virtual functions are overridden by functions that have the same signature and are defined in subclasses. Since the names of constructors and destructors involve the names of their different classes, it would seem that constructors and destructors could not be declared virtual. That is indeed true for constructors. However, an exception is made for destructors.

Every class has a unique destructor, either defined explicitly within the class definition or implicitly by the compiler. An explicit destructor may be defined to be virtual. The following example illustrates the value in defining a virtual destructor:

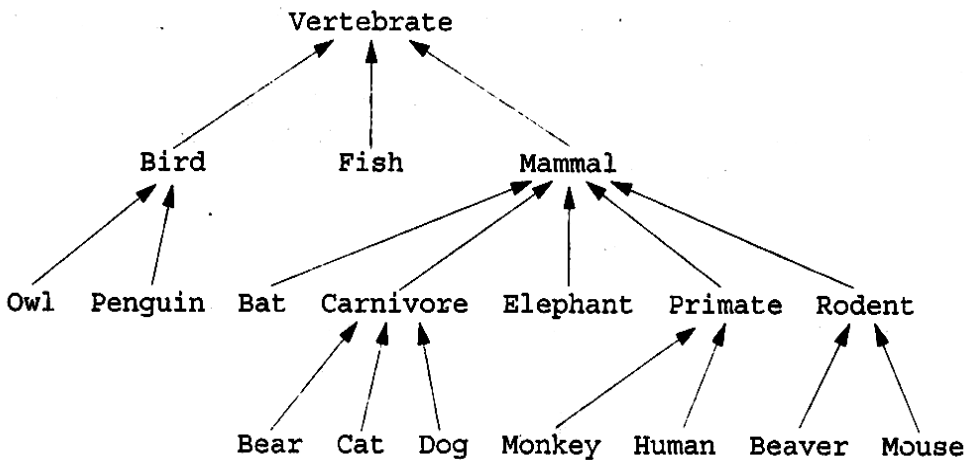
ABSTRACT BASE CLASSES

A well-designed object-oriented program will include a hierarchy of classes whose interrelationships can be described by a tree diagram like the one below. The classes at the leaves of this tree (e.g., Owl, Fish, Dog) would include specific functions that implement the behavior of their respective classes (e.g., Fish.swim ( ), Owl.fly ( ), Dog.dig ( )). However, some of these functions may be common to all the subclasses of

a class (e.g., Vertebrate.eat ( ), Mammal.suckle ( ), Primate.peel ( )). Such functions are likely to be declared virtual in these base classes, and then overridden in their subclasses for specific implementations.

If a virtual function is certain to be overridden in all of its subclasses, then there is no need to implement it at all in its base class. This is done by making the virtual function "pure." A pure virtual member function is a virtual function that has no implementation in its class. The syntax for specifying a pure virtual member function is to insert the initializer "= 0;" in place of the functions body, like this:

```
virtual int f( ) = 0;
```



For example, in the vertebrate class above, we might decide that the eat ( ) function would be overridden in every one of its subclasses, and thus declare it as a pure virtual member function within its Vertebrate base class:

```
class Vertebrate
{    public:
        virtual void eat( ) = 0;    // pure virtual function
};
class Fish : public Vertebrate
```

```
{    public:
        void eat( );                    // implemented specifically for Fish
    class elsewhere
};
```

The individual classes in a class hierarchy are designated as either "abstract" or "concrete" according to whether they have any pure virtual member functions. An abstract base class is a class that has one or more pure virtual member functions. A concrete derived class is a class that does not have any pure virtual member functions. In the example above, the Vertebrate class is an abstract base class, and the Fish class is a concrete derived class. Abstract base classes cannot be instantiated.

The existence of a pure .virtual member function in a class requires that every one of its concrete derived subclasses implement the function. In the example above, if the methods Vertebrate.eat ( ), Mammal.suckle ( ), and Primate.peel ( ) were the only pure virtual functions, then the abstract base classes ("ABCs") would be Vertebrate, Mammal, and Primate, and the other 15 classes would be concrete derived classes ("CDCs"). Each of these 15 CDCs would have its own implementation of the eat ( ) function, the 11 CDCs of the Mammal class would have their own implementation of the suckle ( ) function, and the 2 CDCs of the Primate class would have their own implementation of the peel ( ) function.

An ABC is typically defined during the first stages of the process of developing a class hierarchy. It lays out the framework from which the details are derived in the ABC's subclasses. Its pure virtual functions prescribe a certain uniformity within the hierarchy.

Static members of a C++ class

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the

following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

Let us try the following example to understand the concept of static data members:

```
#include <iostream>

using namespace std;

class Box {
   public:
      static int objectCount;
      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // Increase every time object is created
         objectCount++;
      }

      double Volume() {
         return length * breadth * height;
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;

   return 0;
```

```
}
```

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

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members:

```
#include <iostream>

using namespace std;

class Box {
  public:
    static int objectCount;
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
      // Increase every time object is created
      objectCount++;
    }

    double Volume() {
      return length * breadth * height;
```

```
      }

      static int getCount() {
         return objectCount;
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {

   // Print total number of objects before creating object.
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() << endl;

   return 0;
}
```

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

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

Graded Questions:-

Basic

1. Define Operator overloading. Explain the set of rules required for overloading operator's in C++.
2. Name the operators that can be overloaded
3. Name the operator that cannot be overloaded
4. Write a short note on this pointer.
5. What are pure virtual functions?
6. What are virtual destrucors?
7. Write a C++ program to overload the assignment operator (=) for assigning one complex number into another.
8. Write a C++ program to overload the operator unary (-) for negating the elements of a matrix.

Intermediate:

1. Define friend function. Explain the concept with an example program.
2. Explain function overloading with example.
3. Describe the concept of overloading the subscript operator with an example program.

4. Explain type conversion? What are the types of it?
5. How are virtual functions implemented?
6. Explain static function with example.
7. What is an abstract class? Explain with example.
8. Explain the overloading of the following operators in detail along with an example
    - Assignment(=)
    - Unary
    - Binary
    - Arithmetic
    - Increment (++)
    - Decrement(--)
    - Relational operator(<,>,=<,=>)
    - Stream operator(<<,>>)
9. Write a C++ program to overload the + and -operator for adding and subtracting two complex numbers

Advanced:

1. Write a friend function for adding the two different subject marks and display its sum, using two classes.

2. Write a C++ program to compare two employee's salary details and returns the details of the employee with highest salary using this pointer. (Employee should maintain the details like empno, empname and empsalary)
3. Explain class type to basic type with an example
4. Explain base type to class type with an example
5. Write a C++ program to overload the relational operator(>) to compare two employees salary details and display the details of the employee with highest salary(Employee should maintain the details like empno, empname and empsalary)
6. Write a C++ program to overload the increment, decrement and unary(~) operator for a complex number to increment, decrement and negate its real and imaginary part.

Multiple Choice Questions :-

1. Which of the following perfect set of operators can't be overloaded in CPP ?

a. +=, ?, :: , >>

b. >>, <<, ?, *, sizeof()

c. :: , . , .* , ?:

d. :: , ->, * , new, delete

2. When overloading unary operators using Friend function, it requires_____ argument/s.

a. Zero

b. One

c. Two

d. None of these.

3. While overloading binary operators using member function, it requires ___ argument/s.

a. Zero

b. One

c. Two

d. Three

4. In case of operator overloading, operator function must be_____

1. Static member functions

2. Non- static member functions

3. Friend Functions

a. Only 2

b. Only 1, 3

c. Only 2 , 3

d. All 1 , 2, 3

5. In operator overloading, the overloaded operators must have at least _____ operand of user-defined type.

a. Zero

b. One

c. Two

d. None of these

6. An operator function is created using _____ keyword.

a. iterator

b. allocator

c. constructor

d. operator

7. In case of binary operator overloading with member function, which of following statement should be taken into consideration?

a. Right hand operand must be object

b. Left hand operand must be object

c. Both the operands must be objects

d. All of these should be considered

8. Which of the following statements is NOT valid about operator overloading?

a) Only existing operators can be overloaded.

b) Overloaded operator must have at least one operand of its class type.
c) The overloaded operators follow the syntax rules of the original operator.
d) None of the mentioned

9. Operator overloading is

a) making c++ operator works with objects

b) giving new meaning to existing operator

c) making new operator

d) both a & b

10. What is the return type of the conversion operator?

a) void

b) int

c) float

d) no return type

11. How many parameters does a conversion operator may take?
a) 0

b) 1

c) 2

d) as many as possible

12. Pick out the correct syntax of operator conversion.

a) operator float()const

b) operator float()

c) operator const

d) None of the mentioned

13. Which operator is having the highest precedence?

a) postfix

b) unary

c) shift

d) equality

14. What is this operator called ?: ?

a) conditional

b) relational

c) casting operator

d) none of the mentioned

15) Which class is used to design the base class?

a) **abstract class**

b) derived class

c) base class

d) None of the mentioned

16) Which is used to create a pure virtual function ?

a) $

b) **=0**

c) &

d) !

17) Which is also called as abstract class?

a) virtual function

b) **pure virtual function**

c) derived class

d) None of the mentioned

18) What is meant by pure virtual function?

a) **Function which does not have definition of its own.**

b) Function which does have definition of its own.

c) Function which does not have any return type.

d) None of the mentioned