Operator Overloading

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. We have stated more than once that C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C + + permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operator's with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as *operator overloading*.

Operator overloading provides a flexible option for the creation of new definition for most of the C++ operators. We can create a new language of our own by the creative use of the function and operator overloading techniques. We can overload (give additional meaning to) all the C++ operators except the following:

- Class member access operators (. , .*).
- Scope resolution operator (::).
* Size operator (sizeof).
* Conditional operator (?:)

The excluded operators are very few when compared to the large number of operators which quality for the operator overloading definition.

Although the semantics of an operator can be extended, we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence associatively. For example, the multiplication operator will enjoy higher precedence than the addition operator. Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors can still be used to add two integers.

Defining 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 *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 or friend function. 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 argument 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.

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

```
#include <iostream.h>

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.display( );

    -S;

    cout « “S : “;
    S.display( );
    return 0;
}
```

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.

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

```
friend void  operator-(space &S)
{
        S.x = - S.x;
        S.y = -S.y;
        S.z = -S.z;
}
```

Overloading Binary Operators

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. We know how to add two complex using a friend function. A statement like

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

can be 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 following Program shows how this is accomplished.

```
#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.y;
        return(temp);

}

void complex :: display(void)
{
        cout « x « " + " « y « "\n";
}
```

```
int main( )
{
        complex Cl, C2, C3;
    Cl = compiex(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( );
}
```

Overloading Binary Operators Using Friends

As stated earlier, friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the above section can be modified using a friend operator function as follows:

1. Replace the member function declaration by the friend function declaration.

```
friend complex operator+(complex,   complex);
```

2. Redefine the operator function as follows

```
complex operator+(complex a,  complex b)
{
        return   complex ((a.x+b.x),(a.y+b.y));
}
```

Manipulation of string using operator overloading.

We shall be able to use statements like

```
string s = stringl + string2;

if(stringl >= string2)  string = string1;
```

Strings can be defined as class objects which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use new to allocate memory for each string and a pointer variable to point to the string array. Thus we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations.

A typical string class will look as follows:

```
#include <string.h>
#include <iostream.h>
class string
{
        char *p;
        int  Ten;
        public:
            string( )
            {
                    len = 0;
                    p = 0;
            }
            string(const char * s);
            friend string operator+(const string &s,  const string &t);
            stringg(const string & s);
            ~ string( )
            {
                    delete p;
            }
            friend int operator <= (const string &s, const string &t);
            friend void show(const string s);
};

string  ::  string(const char *s)
 {
        len  = strlen(s);
        p     = new char[1en+1] ;
        strcpy(p,s);
}

string   ::   string (const string & s)
{
        len  =  s.len;
        p     = new char[len+1] ;
        strcpy(p, s.p);
}
```

```
string operator+(const  string &s,  const  string &t) // overloading + operator
{
        string  temp;
        temp.len = s.len + t.len;
        temp.p = new char[temp.len+1];
        strcpy(temp.p,s.p);
        strcat(temp.p,t.p);
        return(temp);
}

int  operator <= (const string &s, const string &t) //   overloading <= operator
{
        int m = strlen(s.p);
        int n = strlen(t.p);
        if(m <=  n)
          return(1);
        else
         return(0);
}

void show(const  string s)
{
        cout  « s.p;
}

int main( )
{
        string s1 = "New ";
        string s2 = "York";
        string s3 = i:Delhi";

        string tl,t2,t3;

        tl = s1;
        t2 = s2;
        t3 = sl+s3;

        cout «     "\ntl = "; show(tl);
        cout «     "\nt2 = "; show(t2);

        cout «     "\n";
        cout «     "\nt3 - "; show(t3);
        cout «     "\n\n,";
```

```
        if (t1 <= t3)
        {
            show(tl);
            cout « " smaller than  ";
            show(t3);
            cout  « "\n";
        }
        else
        {
            show(t3);
            cout « " smaller than ";
            show(tl);
            cout « "\n";
        }

        return 0;
}
```

The following is the output of program

t1 = New
t2 = York
t3 = New Delhi

New Smaller than New Delhi

Rules For Operator Overloading

Although it looks simple to redefine the operators, there are certain restriction and limitation in, overloading them. Some of them are listed below:

1 Only existing operators can be overloaded. New operators cannot be created.

2. The overloaded operator must have at least one operand that is of user-defined type

3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.

4. Overloaded operators follow the syntax rules of the original operators. They can not be overridden.

5. There are some operators that cannot be overloaded.

6. We cannot use friend functions to overload certain operators. However member functions can be used to overload them

7. Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values, but, those overloaded by means of a friend function take one reference argument of the object of the relevant class.

8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

10. Unary arithmetic operators such as +, -, *, and / must explicitly return a value. They must not attempt to change their own arguments

Operators That cannot be overloaded

sizeof	Size of operator
.	Membership operator.
.*	Pointer-to-member operator
::	Scope resolution operator
? :	Conditional operator

Where a friend cannot be used

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

Type Conversions

We know that when constants and variables of different types are mixed in expression, C applies automatic type conversion to the operands as pet certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left.

Consider the following statement that adds two objects and then assigns the third object.

```
v3 = v1+ v2;
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, that the values of all the data members of the right-hand objects are simply copied into the corresponding

members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable? Or. What if they belong to two different classes?

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. We must, therefore design the conversion routines by ourselves, if such operations are required.

Three types of situations might arise in the data conversion between incompatible type.

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

We shall discuss all the three cases in detail.

Basic to Class Type

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an int type arrays. Similarly we used another constructor to build a string type object from a char* type variable. These are all examples where constructors perform a *defacto* type conversion from the argument type to the constructor's class type.

Consider the following constructor:

```
string :: string(char *a)

{
   length = strlen(a);
   P = new char[lengtn+l];
   strcpy(P,a);
}
```

This constructor builds a string type object from a char* type variable a. The variable length and p are data members of the class string. Once this constructor has been defined in the string class, it can be used for conversion from char* type to string type. Example

```
String S1, S2;
Char *name1 = “IBM PC”;
Char *name2 = “APPLE COMPUTERS”;
```

```
S1 = string(nam1);
S2 = name;
```

Class to Basic Type

The constructors did a fine job in type conversion from a basic to class type. What about the conversion from a class to basic type? The constructor functions do not support this operation. C++ allows us to define an overloaded *casting operator* that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function usually referred to as a Conversion function. is:

```
Operator typename( )
{
    ... ...
    ... ...   (function statements)

}
```

This function converts a class type data to typename. For example, the operator double( ) converts a class to type double, the operator int( ) converts a class type object to typeint, and so on.

```
Vector :: operator double( )
{
    double sum = 0;
    for (int I =0; I < size; i++)
            sum = sum + v[i] * v[i];
    return sqrt(sum);
}
```

```
double length = double(v1); OR
double length = V1;
```

The casting operator function should satisfy the following conditions.

1. It must be a class member.
2. It must not specify a return type.
3. It must not have any arguments.

One class to another Class type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But there are situation where we would like to convert one class type data to another class type.

```
objX = objY ;
```

objX is an object of class X and objY is an object of class Y. the class Y type data is converted to class X type data and the converted value is assigned to the objX.

Since the conversion takes place from class Y to class X , Y is known as the source class and X is known as the destination class.

This conversion can be accomplished by using either the constructor function or operator type function. But both should not be used at a time because it creates ambiguity to compiler.