Chapter VIII

Operator Overloading

Operator overloading simplifies the code needed for object manipulation. For example instead of writing A.add(B) where A and B are instances of a class Matrix one can simply write A+B. This simplifies extensively programming using classes. Being able to extend the use of most operator to objects gives C++ its **extensibility** feature. That is, C++ has the power of extending the language to include new types. For example, Matrices are not variables which are part of the language but because of the extensibility feature of C++ one can add Matrix as new type and manipulate matrices as regular built-in variable.

8.1 Rules of Operator Overloading

- You can overload any of the following operators.

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

- You cannot extend the language by inventing new operators. You must limit yourself to existing operators.

  e.g. a**2 // illegal ** is not an existing operator
- You cannot change an operator precedence. For example, the multiplication operator has a higher precedence than the addition operator, so that multiplication is formed first as in:

  a = b + c*d; // same as a = b + (c*d);
- You cannot change an operator's associativity. For example, the addition and subtraction operators are both left associative, so the following expression is evaluated from left to right

  a = b + c - d; // same as a = (b + c) - d;
- You cannot change the way an operator works with built-in types. For example, you cannot change the meaning of the + operator for integers.
- You cannot overload the following operators:

.	class member operator
.*	pointer to member operator
::	scope resolution operator
?:	conditional expression operator

The syntax for overloading an operator
Class Name & **operator** *a valid operator*(*Class Name* & *variable*)
e.g. _Complex& **operator** += (_Complex c);   // overloading +=

Note that for binary operators, (e.g. +, -, *, etc.), there is only one argument in operator. For unary operators (e.g. negation -, ~, !) no arguments are allowed. The compiler interprets c1+=c2 as c1.operator+=(c2); where c2 is the argument placed between the brackets in the statement: _Complex& **operator** += (_Complex c).

To allow more than one argument when a overloading binary operator we use operator as a **friend** function. A friend function is not a member function but is allowed to access all private members of the befriended class.

To illustrate operator overloading and friend functions we will develop a class for complex numbers that allows assignment, addition, etc of complex values.

```
#include <iostream.h>
#include <conio.h>

class _Complex
   {
   private:
      double Real, Imag;
   public:
      _Complex(double realval=0, double imagval=0)
         {
         assign(realval, imagval);
         }

       _Complex(_Complex &c);  //copy constructor

       double getReal()  const
         {
         return Real;
```

```
  }

double getImag() const
  {
  return Imag;
  }

void assign(double realval=0, double imagval=0)
 {
 Real=realval;
 Imag=imagval;
 }

//overloading assignment operator =
_Complex&  operator = (_Complex c);

//overloading +=
_Complex& operator +=(_Complex c);

//overloading +
// the argument c2 is considered as the second operand
// i.e. c1+c2
_Complex& operator + (_Complex c2);
// Note that c1+c2 is interpreted by the compiler as
// c1.operator+(c2)

//friend function
//allows addition of double + _Complex
friend _Complex& operator + (double c1, _Complex c2);

//allows addition of _Complex + double
friend _Complex& operator + (_Complex c1, double c2);

//output stream
// overloading <<
friend ostream& operator << (ostream &os, _Complex c);

//input stream
// overloading >>
friend istream& operator >> (istream &is, _Complex &c);
   }; // end of class definition
```

```
_Complex::_Complex(_Complex &c) //Copy constructor
   {
   Real=c.Real;
   Imag=c.Imag;
     }

_Complex& _Complex::operator = (_Complex c)
    {
    Real = c.Real;
    Imag = c.Imag;
    return *this;
    }


_Complex& _Complex::operator += (_Complex c)
   {
   Real += c.Real;
   Imag += c.Imag;
   return *this;
   }

_Complex _result;  // variable declared outside the proceeding functions

_Complex& _Complex::operator + (_Complex c2)
  {
  double r=Real + c2.Real;
  double i=Imag + c2.Imag;
  _result = _Complex(r,i);
  return _result; //note that local variables cannot be
     }       //returned by reference - hence _result is
            // declared external to the function

_Complex& operator + (double c1, _Complex c2)
  {
  double r = c2.Real + c1;
  double i = c2.Imag;
  _result = _Complex(r,i);
  return _result;
  }

_Complex& operator + (_Complex c1, double c2)
  {
  double r = c1.Real + c2;
```

```
    double i = c1.Imag;
    _result = _Complex(r,i);
    return _result;
    }

ostream& operator << (ostream& os, _Complex c)
 {
 if(c.Imag > 0)
   os << c.Real << " + i" << c.Imag;
 else
   os << c.Real << " - i" << -c.Imag;
 return os;
 }

istream& operator >> (istream& is, _Complex& c)
 {
 is >> c.Real >> c.Imag;
 return is;
  }

//-----------------------------------------------------------------------

int main()
{
_Complex c1(4,5);
_Complex c2(8,-9);
_Complex c3;
_Complex c4(3,7);

c3=c1+c2;
cout << c3 << endl;

c4+=c3;
cout << c4 << endl;

c3=5+c1;
cout << c3 << endl;

cout << "Enter a complex number: --> ";
cin >> c3;
cout << c3 << endl;

getch();
```

```
return 1;
  }
```

The above class overloads the =, +, +=, <<, >> operators. You can of course, as an exercise, extend the class to include division and multiplication. Two friend functions are used to allow the addition of double with _Complex. These, however, will not be needed if we have a function for conversion of double to complex and vice versa. We will cover that topic after the next section. One other thing to notice is that data representation is internal to the class. For example, the print-out of complex numbers is carried-out by a member function. This characteristic of C++ is called **data abstraction**. Data abstraction goes beyond just simple printing of data. For example, data can be stored in a special form that is totally transparent to the user of the class. This form can be modified in the future by the developer of the class without causing error in programs developed under previous versions of the class.

8.2 Overloading operators for an Array

Before we study this topic we will first re-examine return by reference.

Functions Calls on the Left of the equal sign

Consider the following program:

```
#include <iostream.h>
#include <conio.h>

int x;

int& getx()
  {
  return x;
  }

//-----------------------------------------------------------------------

int main()
{

getx()=4;   //same as x=4;
cout << " x= " << x << endl;
cout << " getx= " << x << endl;
getch();
 return 1;
   }
```

```
Output:

 x= 4
 getx= 4
```

From the above program one can conclude that a function that returns a reference can be treated as if it were a variable. We will use that feature in the next topic.

Overloading the [ ] Operator for Arrays

The following program illustrates an example in which the [] is overloaded. We have utilized the fact that a function having a return by reference can appear on the left-hand side and accepts values in the main program in the statement a[i]=double(i);

```
#include <iostream.h>
#include <conio.h>

const unsigned MIN_SIZE=5;
const double BAD_VALUE=-1.0e30;

class Array
  {
  private:
    double *dataPtr;
    unsigned size;
    double badIndex;

  public:
    Array(unsigned length = MIN_SIZE);
    ~Array()
      {
      delete [ ] dataPtr;
      }
    unsigned getSize() const
     {
     return size;
     }

    double& operator [ ](unsigned index);
    };

Array::Array(unsigned length)
 {
```

```
 unsigned i;
 size=(length < MIN_SIZE)? MIN_SIZE:length;
 badIndex=BAD_VALUE;
 dataPtr=new double[size];
 for(i=0; i<size; i++)
   *(dataPtr+i)=0.0;
   }

 double& Array::operator [ ](unsigned index)
  {
  if(index < size)
    return *(dataPtr + index);
  else
    return badIndex;
    }

//--------------------------------------------------------------------------

int main()
{
Array a(12);
int i;
int n=a.getSize();

for(i=0; i<n; i++)
   a[i]=double(i);

for(i=0; i< n+2; i++)
  cout << a[i] << endl;

//Using a single-dimensional array to store
// 2-D data (matrix)

int j, NR, NC;
NR=3; NC=4;

for(i=0; i<NR; i++)
  for(j=0; j<NC; j++)
   a[j+i*NC]=j+i*NC;

for(i=0; i<NR; i++)
  {
  cout << endl;
  for(j=0; j<NC; j++)
    cout << a[j+i*NC] << ' ';
    }
cout << endl;
getch();
 return 1;
   }
```

```
Print out
0
1
2
3
4
5
6
7
8
9
10
11
-1e+30
-1e+30

0 1 2 3
4 5 6 7
8 9 10 11
```

The above also illustrates the use of a 1-D array in storing a 2-D array. As a matter of fact a 1-D array can readily store a multidimensional array, this is how arrays of any dimension are stored in memory.

8.3 Conversion Operators

In C++ you can define conversions between classes, or between a class and a built-in type.

Conversion by Constructor

The constructor in the class _Complex is:

Complex(**double** realval=0, **double** imagval=0);

converts _Complex(6) to 6+i0.

Conversion operator

The syntax of the conversion member from class type to double is:

operator double( ) **const**;

Note that it accepts no return type and no arguments.
As example we will add the conversion type in _Complex and remove the two friend functions for adding double to _Complex and _Complex to double.

```
#include <iostream.h>
#include <conio.h>

class _Complex
   {
   private:
      double Real, Imag;
   public:
      _Complex(double realval=0, double imagval=0)
         {
         assign(realval, imagval);
         }

       _Complex(_Complex &c);  //copy constructor

       double getReal()  const
        {
        return Real;
        }

       double getImag() const
        {
        return Imag;
        }

       void assign(double realval=0, double imagval=0)
        {
        Real=realval;
        Imag=imagval;
        }

       //overloading assignment operator =
       _Complex&  operator = (_Complex c);

       //overloading +=
       _Complex& operator +=(_Complex c);
```

```
    //overloading +
    // the argument c2 is considered as the second operand
    // i.e. c1+c2
    _Complex& operator + (_Complex c2);
    // Note that c1+c2 is interpreted by the compiler as
    // c1.operator+(c2)

    //Conversion function from _Complex to double
    operator double () const;

    //output stream
    // overloading <<
    friend ostream& operator << (ostream &os, _Complex c);

    //input stream
    // overloading >>
    friend istream& operator >> (istream &is, _Complex &c);
     }; // end of class definition


 _Complex::_Complex(_Complex &c) //Copy constructor
   {
   Real=c.Real;
   Imag=c.Imag;
    }

_Complex& _Complex::operator = (_Complex c)
    {
    Real = c.Real;
    Imag = c.Imag;
    return *this;
    }

/*_Complex _Complex::operator = (_Complex c)
    {
    Real = c.Real;
    Imag = c.Imag;
    return *this;
    }   */

_Complex& _Complex::operator += (_Complex c)
   {
   Real += c.Real;
```

```
     Imag += c.Imag;
     return *this;
     }

_Complex _result;  // variable declared outside the proceeding functions

_Complex& _Complex::operator + (_Complex c2)
    {
   double r=Real + c2.Real;
   double i=Imag + c2.Imag;
   _result = _Complex(r,i);
   return _result; //note that local variables cannot be
       }        //returned by reference - hence _result is
                // declared external to the function

_Complex::operator double() const
   {
   return (double)Real;
   }

ostream& operator << (ostream& os, _Complex c)
 {
 if(c.Imag > 0)
   os << c.Real << " + i" << c.Imag;
 else
   os << c.Real << " - i" << -c.Imag;
 return os;
 }

istream& operator >> (istream& is, _Complex& c)
 {
 is >> c.Real >> c.Imag;
 return is;
  }



//------------------------------------------------------------------------

int main()
{
_Complex c1(4,5);
_Complex c2(8,-9);
_Complex c3;
_Complex c4(3,7);

c3=c1+c2;
cout << c3 << endl;

c4+=c3;
cout << c4 << endl;

c3=_Complex(5)+c1;
 //The _Complex(5) to prevent the conversion of c1 to double
cout << c3 << endl;

double a=c3; //uses conversion function
cout << a << endl;

cout << "Enter a complex number: --> ";
cin >> c3;
cout << c3 << endl;

getch();
 return 1;
   }
```

Note that in the main program we explicitly had to convert 5 to _Complex in c3=_Complex(5)+c1, by using the constructor member for the class.

To convert from a user defined class to a user defined class the syntax is the same as from user-defined to built-in.

Problems

2. Extend the class _Complex developed in this chapter to include multiplication, division, conjugate value, absolute value of _Complex numbers.
3. The round brackets, ( ), can be overloaded to take on two arguments. Develop a C++ class for handling matrices and overloads the round brackets for assigning a value to a matrix element or reading a value of a matrix element.
4. Using the class _Complex develop C++ function for: e^{ix}, $\sin(x)$, $\cos(x)$ where x is real and

$$e^{ix} = 1 + (ix) + \frac{(ix)^2}{2!} + \frac{(ix)^3}{3!} + \frac{(ix)^4}{4!} + \ldots.$$

$$\sin(x) = \frac{e^{ix} - e^{-ix}}{2i}$$

$$\cos(x) = \frac{e^{ix} + e^{-ix}}{2}$$

5. Develop a class ComplexMatrix to handle matrices with complex values. Include a member function for solving a set of linear algebraic equations with complex coefficients.
6. Include member functions in the class developed in question 4 for converting a matrix of complex type to one of real type by assigning the real values of the complex matrix to the real matrix.
7. Extend question 4 to deal with the exponent, sine and cosine of a matrix.