OPERATOR OVERLOADING AND TYPE CONVERSIONS

INTRODUCTION

The mechanism of giving additional meanings to an operator is known as **operator overloading**.

DEFINING OPERATOR OVERLOADING

An additional task to an operator can be defined with the help of a special function, called **operator functions**. The general form of an operator function is:

```
return-type class-name :: operator op(argument-list)
 {
        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 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. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototypes as follows:

```
complex operator+(complex);
complex operator-(complex);
friend complex operator+(complex,complex);
friend complex operator-(complex,complex);
complex operator==(complex);
friend complex operator<(complex,complex);
```

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

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.

OVERLOADING UNARY OPERATORS

A unary operators act on only one operand. Example of unary operators are the increment operator ++ , decrement operator --, unary minus, etc. The unary minus operator changes the sign of an operand when applied to a basic data item. The unary minus when applied to an object should change the sign of each of its data items. The following program shows how the unary minus operator is overloaded.

```
#include<iostream.h>
class unaryminus
 {
        int x,y,x;
    public:
        void getdata();
        void operator-();
        void display();
 };

void unaryminus::getdata()
 {
        cout<<"Enter the values : ";
        cin>>x>>y>>z;
 }
void unaryminus::operator-()
 {
        x=-x;
        y=-y;
        z=-z;
 }
void unaryminus::display()
 {
        cout<<"\nx="<<x;
        cout<<"\ny="<<y;
        cout<<"\nz="<<z;
}
void main()
 {
        unaryminus um;
        um.getdata();
        um.display();
        -um;
        um.display();
 }
```

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

```
friend void operator-(unaryminus &um);          //declaration

void operator-(unaryminus &um)                  //definition
{
        um.x=-um.x;
        um.y=-um.y;
        um.z=-um.z;
}
```

OVERLOADING BINARY OPERATORS

Binary operators overloaded by means of member functions take one formal argument which is the value to the right of the operator. Binary operators require two operands to perform the operation. The following program shows add two complex numbers by overloading binary operator +.

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 arugment.

```
#include<iostream.h>
class complex
{
        int i,r;
    public:
        void getdata();
        complex operator+(complex);
        void display();
};

void complex::getdata()
{
        cout<<"Enter the complex number : ";
        cin>>r>>i;
}

complex complex::operator+(complex c)
{
        complex t;
        t.r=r+c.r;
        t.i=i+c.i;
        return(t);
}
void complex::display()
{
        if (i<0)
                cout<<"\n"<r<<i<<"i";
        else
                cout<<"\n"<r<<"+"<<i<<"i";
}
```

```
void main()
 {
        complex c1,c2,c3;
        c1.getdata();
        c2.getdata();
        c3=c1+c2;               //c3=c1.operator+(c2);
        c1.display();
        c2.display();
        c3.display();
 }
```

OVERLOADING BINARY OPERATORS USING FRIENDS

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.

```
#include<iostream.h>
class complex
 {
        int i,r;
    public:
        void getdata();
        friend complex operator+(complex, complex);
        void display();
 };

void complex::getdata()
 {
        cout<<"Enter the complex number : ";
        cin>>r>>i;
 }

complex complex::operator+(complex c1, complex c2)
 {
        complex t;
        t.r=c1.r+c2.r;
        t.i=c1.i+c2.i;
        return(t);
 }

void complex::display()
 {
        if (i<0)
                cout<<"\n"<r<<i<<"i";
        else
                cout<<"\n"<r<<"+"<<i<<"i";
}
```

```
void main()
 {
        complex c1,c2,c3;
        c1.getdata();
        c2.getdata();
        c3=c1+c2;                       //c3=operator+(c1,c2);
        c1.display();
        c2.display();
        c3.display();
 }
```

OVERLOADING INSERTION (<<) AND EXTRACTION (>>) OPERATORS

We can use a friend function with a built-in type data as the left-hand operand and an object as the right-hand operand. The following program illustrates this, using scalar multiplication of a vector. It also shows how to overload the input and output operators>> and <<.

```
#include<iostream.h>
const size = 3;
class vector
 {
        int v[size];
    public:
        vector();
        vector(int *x);
        friend vector operator *(int a, vector b);
        friend vector operator *(vector b, int a);
        friend istream & operator >> (istream &, vector &);
        friend ostream & operator << (ostream &, vector &);
 };

vector:: vector()
 {
        for(int i=0;i<size;i++)
                v[i]=0;
 }
vector:: vector(int *x)
 {
        for(int i=0;i<size;i++)
                v[i]=x[i];
 }

vector operator*(int a, vector b)
 {
        vector c;
        for(int i=0;i<size;i++)
                c.v[i]=a*b.v[i];
        return c;
 }
```

```
vector operator*(vector b, int a)
{
        vector c;
        for(int i=0;i<size;i++)
                c.v[i]=b.v[i]*a;
        return c;
}

istream &operator >>(istream &din,vector &b)
{
        for(int i=0;i<size;i++)
                din>>b.v[i];
        return (din);
}

ostream &operator <<(ostream &dout,vector &b)
{
        dout<<"("<<b.v[0];
        for(int i=1;i<size;i++)
                dout<<","<<b.v[i];
        return (dout);
}

int x[size]={2,4,6};

void main()
{
        vector m;                       //invokes constructor 1
        vector n=x;                     //invokes constructor 2
        cout<<"Enter elements of vector m\n";
        cin>>m;                         //invokes operator>>() function
        cout<<"\nm="<<m;                //invokes operator<<() function
        vector p,q;
        p=2*m;                          //invokes friend 1
        q=n*2;                          //invokes friend 2
        cout<<"\np="<<p;                //invokes operator<<()
        cout<<"\nq="<<q;                //invokes operator<<()
}
```

MANIPULATION OF STRINGS USING OPERATORS

C++ permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers for example, we shall be able to use statements like,

```
string3=string1+string2;
if (string1>=string2) string=string1;
```

strings can be defined as class objects which can be the manipulated like the built-in types. A typical string class will look as follows:

```
class string
{
        char *p;
        int len;
   public:
        …..
        …..
        …..
};
```

We shall consider an example to illustrate the application of overloaded operators to strings. The following example overloads two operators + and == just to show how they are implemented.

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

class string
 {
        char str[80];
    public:
        string()
         {
                strcpy(str," ");
         }

        string(char s[])
         {
                strcpy(str,s);
         }

        void display()
         {
                cout<<"\n"<<str);
         }

        string operator+(string s)
         {
                string t;
                strcpy(t.str,str);
                strcat(t.str,s.str);
                return t;
         }

        int operator==(string s)
         {
                return(((strcmp(str,s.str)==0)?1:0);
         }
};
```

```
void main()
 {
        string s1="C++";
        string s2=" VC++";
        string s3;
        s3=s1+s2;
        s1.display();
        s2.display();
        s3.display();
        if (s1==s2)
                cout<<"\nThe strings are equal";
        else
                cout<<"\nThe strings are not equal";
 }
```

RULES FOR OVERLOADING OPERATORS

There are certain restrictions and limitations in overloading operators. Some of them are listed below.

1. Only existing operators can be overloaded.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We can not change the basic meaning of an operator. That is, we can not redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators.
5. Overloaded operators can not be overridden.
6. There are some operators that can not be overloaded. (shown below)

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

7. We can not use friend functions to overload the following operators.

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

8. Unary operators, overloaded by means of a member function, take no explicit argument.
9. Binary operators overloaded through a member function take one explicit argument.
10. Binary arithmetic operators such as +, -, * and / must explicitly return a value.

Overloading Unary Operator ++

```
#include <iostream>
using namespace std;
class increment
{

int a;
public:
   increment(){a=0;}
   increment operator++()       //prefix
   {
      a++;
      cout<<a;
      return *this;
   }
   increment operator++(int)  // postfix
   {
      a++;
      cout<<a;
      return *this;
   }
};

int main()
{
   increment i;
   increment x=++i;
   increment y=i++;
   ++i;
}
```

DATA TYPE CONVERSION

C notation

(int) float_expression

C++ notation

int(float_expression)

Types of data conversion:

1. Conversion of Built-in Data types to Objects
2. Conversion of Objects to Built-in Data types
3. Conversion of Objects of One type to Another type

Conversion of Built-in Data types to Objects

```
#include<iostream>
using namespace std;
class length
{
   int foot;
   int inch;
public:
   length(int x)
   {
      foot=x/12;
      inch=x%12;
   }
   void display()
   {
      cout<<"\nFoot : "<<foot;
      cout<<"  Inch : "<<inch;
   }
};

int main()
{
   length L1(109);
   L1.display();
   length L2(224);
   L2.display();
   return(0);
}
```

Conversion of Objects to Built-in Data types

```
#include<iostream>
using namespace std;
class length
{
   int foot;
   int inch;
public:
   length(int f,int i)
   {
      foot=f;
      inch=i;
   }
   void display()
   {
      cout<<"\nFoot : "<<foot;
      cout<<"  Inch : "<<inch;
   }
   operator int()
   {
      int var;
      var=foot*12+inch;
      return var;
   }
};
int main()
{
   length L1(9,1);
   cout<<"\nLength in inches1 = "<<L1;
   length L2(18,8);
   cout<<"\nLength in inches2 = "<<L2;
   return(0);
}
```

Conversion of Objects of one type to another type

Length in FPS Units to MKS Units

FPS → Foot, Pound and Second
MKS → Meter, Kilogram, Second

```
#include<iostream>
using namespace std;
class mks
{  double cm;
   double mm;
public:
   mks()
   {
      cm=mm=0;
   }
```

```
   mks(int c,int m)
   {
      cm=c;
      mm=m;
   }
   void display()
   {
      cout<<"\n Centimeter = "<<cm;
      cout<<"   Millimeter = "<<mm;
   }
};

class fps
{
   int foot;
   int inch;
public:
   fps()
   {
      foot=inch=0;
   }fps(int f, int i)
   {
      foot=f;
      inch=i;
   }
   void display()
   {
      cout<<"\n Foot = "<<foot;
      cout<<"   Inch = "<<inch;
   }
   operator mks()
   {
      double x;
      int y,z;
      x=(foot*12+inch)*25.4;
      y=int(x)/10;
      z=int(x)%10;
      return mks(y,z);
   }
};

int main()
{
   fps L1(10,9);
   L1.display();
   mks L2;
   L2=L1;
   L2.display();
   return(0);
}
```

Conversion of data from MKS Units to FPS Units

```
#include<iostream>
using namespace std;
class fps
{
   int foot;
   int inch;
public:
   fps()
   {
      foot=inch=0;
   }

  fps(int f, int i)
   {
      foot=f;
      inch=i;
   }

   void display()
   {
      cout<<"\n Foot = "<<foot;
      cout<<"   Inch = "<<inch;
   }
};

class mks
{
   double cm;
   double mm;
public:

   mks()
   {
      cm=mm=0;
   }

   mks(int c,int m)
   {
      cm=c;
      mm=m;
   }

   void display()
   {
      cout<<"\n Centimeter = "<<cm;
      cout<<"   Millimeter = "<<mm;
   }
```

```
    operator fps()
    {
        int y,z;
        double x,p;
        x=cm*10+mm;
        p=x/25.4;
        y=int(p)/12;
        z=int(p)%12;
        return fps(y,z);
    }
};

int main()
{
    fps L1;
    mks L2(254,0);
    L2.display();
    L1=L2;
    L1.display();
    return(0);
}
```