Operator Overloading

Introduction

- It is one of the many exciting features of C++.

- C++ has ability to provide the operators with a special meaning for a data types.

- We can overload (give additional meaning to) all the C++ operators except:
  - Class member access operators (. & .*)
  - Scope resolution operators (: :)
  - Size operator (sizeof)
  - Conditional operators (? :)

- When an operator is overloaded, its original meaning is not lost.

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 called *operator function*.

```
Return_type class-name : :operator op (arg-list)
{
     Function body   //  task defined
}
```

Defining Operator Overloading

```
Return_type class-name : :operator op (arg-list)
{
     Function body   //  task defined
}
```

- ***Return_type*** is the type of value returned by the specified operation.
- ***op*** is the operator being overloaded.
- ***op*** is preceded by the keyword **operator**.
- ***operator op*** is the function name.

Defining Operator Overloading

continue…

Operator Function must be either

- member function (non-static)

Or

- friend function.

The basic difference :

- A friend function will have only one argument for unary operators and two for binary operators.
- A member function has no arguments for unary operators and one argument 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.
- Arguments may be passed either by value or by reference.

Defining Operator Overloading

continue…

	Unary	Binary
Member function	No arg	1 arg
Friend function	1 arg	2 args

Process of Operator Overloading

The process of overloading involves the following steps:

- Create a class that defines the data type that is to be used in the overloading operation.

- Declare the operator function ***operator op( )*** in the public part of the class. It may be either a member function or a friend function.

- Define the operator function to implement the required operations.

Process of Operator Overloading

Overloaded operator functions can be invoked by expressions such as:

For unary operators: op x or x op

For binary operators: x op y

op x or x op would be interpreted as

for a friend function: operator op (x)

for a member function: x.operator op ( )

x op y would be interpreted as

for a friend function: operator op (x,y)

for a member function: x.operator op (y)

Overloading Unary Operators

Consider a unary minus operator:

- It takes just one operand.
- It 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.

Overloading unary minus

Unary minus operation on built-in types:

```
int a=5;
a = -a;
cout << a;          // Displays –5
```

Unary minus operation on objects:

```
class X {
    public:
    int a;
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
};
```

```
int main( ) {
          X x1(5);
          x1.disp();          // 5

          x1.a=-x1.a;

          x1.disp();          // -5
          return 0;
}
```

Using member function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }

    void operator-() {
            a = -a;
    }
};
```

```
int main( ) {
            X x1(5);
            x1.disp();          // 5

            -x1;
            x1.disp();          // -5
            return 0;
}
```

The compiler interprets –x1 as

x1.operator-();

Using member function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
    X operator-();
};
X  X :: operator-() {
        X temp;
        temp.a = -a;
        return (temp);
}
```

```
int main( ) {
        X x1(5);
        X x2;
        x1.disp();          // 5

        x2 = -x1;
        x2.disp();          // -5
        return 0;
}
```

The compiler interprets x2=–x1 as

x2 = x1.operator-();

Using friend function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
    friend X operator-( X ob);
};
X  operator-( X ob) {
            X temp;
            temp.a = - ob.a;
            return (temp);
}
```

```
int main( ) {
            X x1(5);
            X x2;
            x1.disp();          // 5

            x2 = -x1;
            x2.disp();          // -5
            return 0;
}
```

The compiler interprets x2=–x1 as

x2 = operator-(x1);

Overloading Binary Operators

As a rule, in overloading binary operators,

- the *left-hand* operand is used to invoke the operator function and

- the right-hand operand is passed as an argument.

Overloading binary plus

Binary + operation on built-in types:

```
int a=5,b=10,c;
c = a + b;
cout << c;          // Displays 15
```

Binary + operation on objects:

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
};
```

```
int main( ) {
          X x1(5), x2(10), x3;

          x3.a = x1.a + x2.a;

          x3.disp();          // 15
          return 0;
}
```

Using member function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }

    X operator+( X p) {
        X t;
        t.a = a + p.a;
        return (t);
    }
};
```

```
int main( ) {
            X x1(5),x2(10),x3;

            x3 = x1 + x2;
            x3.disp();          // 15
            return 0;
}
```

- The compiler interprets x3 = x1 + x2 as
  x3 = x1.operator+(x2);

- The first operand always calls the operator function and the second operand is passed as argument

Using friend function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
    friend X operator+( X p,X q);
};
X operator+( X p,X q) {
    X t;
    t.a = p.a + q.a;
    return (t);
    }
```

```
int main( ) {
        X x1(5),x2(10),x3;

        x3 = x1 + x2;
        x3.disp();          // 15
        return 0;
}
```

- The compiler interprets x3 = x1 + x2 as x3 = operator+(x1,x2);

Overloading Binary Operators

```
return  complex((x+c.x), (y+c.y));
```

The compiler invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object.

Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object.

Overloading Binary Operators Using Friends

- Friend function requires two arguments to be explicitly passes to it.
- Member function requires only one.

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

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

Overloading Binary Operators Using Friends

- We can use a friend function with built-in type data as the left-hand operand and an object as the right-hand operand.

Manipulation of Strings using Operators

- There are lot of limitations in string manipulation in C as well as in C++.
- Implementation of strings require character arrays, pointers and string functions.
- C++ permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to other built-in data types.
- ANSI C++ committee has added a new class called string to the C++ class library that supports all kinds of string manipulations.

Manipulation of Strings using Operators

continue…

- Strings can be defined as class objects which can be then manipulated like the built-in types.

- Since the strings vary in size, we use new to allocate memory for each string and a pointer variable to point to the string array.

Manipulation of Strings using Operators

continue…

– We must create string objects that can hold two pieces of information:
    - Length
    - Location

```
class string
{
      char  *p;        // pointer to string
      int    len;      // length of string
   public :
      ------
      ------
};
```

Rules For Overloading Operators

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

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

- We cannot change the basic meaning of an operator.

- Overloaded operators follow the syntax rules of the original operators.

Rules For Overloading Operators

continue…

- The following operators that cannot be overloaded:

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

Rules For Overloading Operators
continue…

- The following operators can be over loaded with the use of member functions and not by the use of friend functions:
  - Assignment operator =
  - Function call operator( )
  - Subscripting operator [ ]
  - Class member access operator ->
- Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument.

Rules For Overloading Operators

continue…

- Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
- Binary arithmetic operators such as +, -, * and / must explicitly return a value. They must not attempt to change their own arguments.

Type Conversions

- The type conversions are automatic only when the data types involved are built-in types.

```
int m;
float x = 3.14159;
m = x; //  convert x to integer before its value is assigned
       //  to m.
```

- For user defined data types, the compiler does not support automatic type conversions.
- We must design the conversion routines by ourselves.

Type Conversions

continue…

Different situations of data conversion between incompatible types.

- Conversion from basic type to class type.
- Conversion from class type to basic type.
- Conversion from one class type to another class type.

Basic to Class Type

A constructor to build a string type object from a char * type variable.

```
string : : string(char *a)
{
  length = strlen(a);
  P = new char[length+1];
  strcpy(P,a);
}
```

The variables length and p are data members of the class string.

Basic to Class Type

continue…

```
string s1, s2;
string name1 = “IBM PC”;
string name2 = “Apple Computers”;
s1 = string(name1);
s2 = name2;
```

First converts name1 from char* type to string type and then assigns the string type value to the object s1.

First converts name2 from char* type to string type and then assigns the string type value to the object s2.

Basic to Class Type

continue…

```
class time
{      int hrs ;
       int mins ;
   public :
      …
   time (int t)
   {
       hrs = t / 60 ;
       mins = t % 60;
   }
} ;

time T1;
int duration = 85;
T1 = duration;
```

Class To Basic Type

A constructor function do not support type conversion from a class type to a basic type.

An overloaded ***casting operator*** is used to convert a class type data to a basic type.

It is also referred to as ***conversion function***.

```
operator typename( )
{
   …
   …  ( function statements )
   …
}
```

This function converts a ***calss type*** data to ***typename***.

Class To Basic Type

continue…

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

This function converts a vector to the square root of the sum of squares of its components.

Class To Basic Type

continue…

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

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

Class To Basic Type

continue…

- Conversion functions are member functions and it is invoked with objects.
- Therefore the values used for conversion inside the function belong to the object that invoked the function.
- This means that the function does not need an argument.

One Class To Another Class Type

objX = objY ; // objects of different types

- **objX** is an object of class **X** and **objY** is an object of class **Y**.
- The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**.
- Conversion is takes place from **class Y** to **class X**.
- **Y** is known as ***source class***.
- **X** is known as ***destination class***.

One Class To Another Class Type

continue…

Conversion between objects of different classes can be carried out by either a constructor or a conversion function.

Choosing of constructor or the conversion function depends upon where we want the type-conversion function to be located in the source class or in the destination class.

One Class To Another Class Type

continue…

operator typename( )

- Converts the class object of which it is a member to typename.
- The typename may be a built-in type or a user-defined one.
- In the case of conversions between objects, typename refers to the destination class.
- When a class needs to be converted, a casting operator function can be used at the source class.
- The conversion takes place in the source class and the result is given to the destination class object.

One Class To Another Class Type

continue…

Consider a constructor function with a single argument

- Construction function will be a member of the destination class.

- The argument belongs to the source class and is passed to the destination class for conversion.

- The conversion constructor be placed in the destination class.