Unit 6 Operator Overloading

Structure:

6.1 Introduction

In the previous unit you have studied the constructors, their different types and their use in C++. You have also studied the destructors and the namespaces in C++. In this unit you study the concept of operator overloading. You will also learn how to overload different operators in C++. You will also know the method of performing type conversions in C++.

Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables. Operator overloading, less commonly known as operator ad-hoc polymorphism, is a specific case of polymorphism, where different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.

Operator overloading is claimed to be useful because it allows the developer to program using notation "closer to the target domain" and allows user-defined types a similar level of syntactic support as types built into the language. In this unit we are going to discuss unary and binary operator overloading with examples.

Objectives:

After studying this unit, you should be able to:

- explain operator overloading
- describe unary operator overloading
- describe binary operator overloading
- discuss type conversion

6.2 Operator Overloading in C++

With the use of operator overloading feature of C++, programmers can specify the way the different arithmetic, relational and other operators work with user defined data types or classes. It allows the programmer to modify the functionality of the existing operators so that they can be used with user defined data types such as classes, in addition to built-in data types. Consider the following statements:

int x,y,z;

z=x+y;

In this example you can observe that the two integer variables are added and the result is stored in the third variable z. But it is not possible in C++ to add the two objects of the class with the same '+' operator. For this purpose you have to use operator overloading concept. For example if obj1, obj2 and obj3 are the three objects of a class the following statements will generate a compile time error if the '+' operator is not overloaded.

obj3= obj1+obj2;

So if you want to perform this operation you have to overload the '+' operator. In C++ almost all the unary, binary and special operators can be overloaded.

In the unit 5, we had written an add function for the class distance. We used a call d3.add (d1,d2) to add two distance objects. It would be easy for us if use the statement d3=d1+d2. But it is only possible if we overload the '+' operator to perform this function.

Shown below is the list of functions that can be overloaded in C.

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

Some operators that cannot be overloaded are:

- dot operator (.)
- dereference pointer to class members (.*)
- scope resolution operator (::)

- *sizeof* operator
- preprocessor symbol (#)

A special function called and *operator function* is used to overload an operator. It defines the operations that the overloaded operator will perform on the objects of the class for which it is redefined. An operator function is defined either as a public function or as a friend function. You must follow the following steps to overload an operator:

- Create the class for which an operator is to be overloaded.
- Declare the function either as a public function or a friend function.
- Define the operator either inside the class definition (member function) or outside the class (friend function).

  The syntax to define the member operator function inside the class is

  ```
  return_type operator op (parameter list)
  {
   //function body
  }
  ```

  Example: int operator + (int a, int b)

  If you are defining member function outside the class you have to declare it first inside the class. The following is the syntax to declare the member function inside the class:

  ```
  return_type operator op (parameter list);
  ```

  The following is the syntax to define member operator function outside the class:

  ```
  return_type class_name:: operator op (parameter_list)
  {
   //function body
  }
  ```

  Example: int test :: operator + (int a, int b)

  Where return_type(int) = data type of the value returned by the function

  operator is C++ keyword

op(+) is the name of the operator to be overloaded as well the name of the operator overloaded function.

parameter_list (int a, int b) is the list of arguments passes to the operator function.

Here class_name is the name of the class to which the object belong

You should keep in mind the following rules while overloading operators.

- An operator function can be either a nonstatic member function, or a nonmember function with at least one parameter that has class, reference to class, enumeration, or reference to enumeration type.
- The precedence and associativity of the operators cannot be changed. However you can use parentheses to change the order of evaluation.
- The number of operators (unary, binary or ternary) in an operator cannot be changed.
- An overloaded operator (except for the function call operator) cannot have default arguments or an ellipsis it is a notation that is used to denote ranges and denoted by series of dots i.e. (.. or …) in the argument list.
- The operators =, [], () and ->can be overloaded only as member functions and not as friend functions.
- Except the assignment operator '=' all the overloaded operators can be inherited by the derived class.

  The general rules discussed in this section are not followed by the following operators – new, delete, new[] and delete[].

Benefits of operator overloading are:
It makes programs easier to read and debug.
It has the ability to provide the operators with a special meaning for a data type.

Drawbacks of operator overloading:
Sometimes you may misunderstand the meaning of overloaded operators (if the natural semantics is not used by the programmer). The extreme case, where the plus-operator is re-defined to mean minus and the minus operator is re-defined to mean plus, probably will not occur very often, but more subtle cases are conceivable. Designing a class library is like designing a

language if you use operator overloading, use it in a uniform manner; do not use it if it can easily give rise to misunderstanding.

Self Assessment Questions

1. The functionality of the operators with any data types can be changed by operator overloading? (True/False).
2. Not all the operators can be overloaded in C++. (True/False).
3. A special function called and _______ function is used to overload an operator.

6.3 Overloading Unary Operators

As you know already that the unary operators are the ones that operate on a single operator. Some of the unary operators are - the increment (++) and decrement (--) operators, the unary minus (-) operator and the logical not (!) operator.

In case of overloading of unary operators, the calling operand can be either left or right side of the operator as in case of increment and decrement operators. While defining the operator functionality for the class the keyword operator is used. Let us overload the increment operator for a class.

```
//unary.cpp
# include <iostream.h>
class counter
{ unsigned int c;
public:
counter()
{c=0;}
int getcount()
{return c;}
void operator ++()
{ c++;}
};
void main()
{ counter c1;
c1++;
++c1;
cout<<c1.getcount();
}
```

In the above example, the operator ++ is an overloaded function. It does not return any value and no arguments are passed in this function. All unary operators do not take arguments as they operate on only one operand and that operand itself invokes the operator. Therefore the operand is sent by default. However the above implementation cannot be used in statements such as c2=c1++; where c1 and c2 are counter variables. The reason being that a void value is returned by the ++ operator and is assigned to the counter variable as shown in the above example. To resolve this problem you can define the operator ++ in a way that it returns the value of the counter variables, as shown below.

```
//increment.cpp
# include <iostream.h>
class counter
{ unsigned int count;
public:
counter()
{c=0;}
counter(int num)
{c=num;}
int getcount()
{return c;}
counter operator ++()
{ c++;
return counter(c);}
};
void main()
{ counter c1,c2;
c1++;
c2=++c1;
cout<<c1.getcount()<<endl;
cout<<c2.getcount();
}
```

In the example shown above you can see that a counter variable is returned without creating a variable. In the above implementation, we are returning the counter variable without creating a variable. It is done by using a constructor having one argument. When the statement return counter(c) is executed, the value of the argument c of the invoking object is passed to the constructor and a nameless object is created which is initialized to the value stored in the count and returned to the calling program. One argument constructor is used in this case. We are using one argument constructor in this case.

The limitation of unary operator overloading is that the increment and decrement operators cannot be totally duplicated. The reason is that the C++ compiler is not able to differentiate between pre and post increment/ decrement operators.

Self Assessment Questions

4. Unary operators are implemented with _____ arguments.
5. Unary operators should have return value as ________.
6. Unary operators overloaded for the class can differentiate between post and pre operators. (True/False)

6.4 Overloading Binary Operators

The operators that require two operands for its operation are known as binary operators. The examples of some of the binary operators are arithmetic operators like +, *, /, etc. relational operators like <,>. In binary operator overloading one operand is the object of the class invoking the function is passed implicitly to the member operator function. And the other operand is passed as an argument, which can be passed either by value or by reference. This concept is explained thorough the example shown below illustrating the overloading of + operator:

```
#include<iostream.h>
class complex
```

```
{
 float x;           //real part
float y;            //imaginary part
public:
complex(){ }    //constructor1
 complex( float real, float imag) //constructor2
 {
   x=real;
   y=imag;
 }
 complex operator+(complex c);
 void display(void);
};
complex complex:: operator+(complex c)
{
 complex temp;           //temporary
 temp.x = x + c.x;
temp.y = y + c.y;
return(temp);
}
void complex :: display(void)
{
 cout << x << "+ j" << y << "\n";
}
int main()
{
 complex c1, c2, c3;              //invokes constructor1
c1= complex(2.5, 3.5);           //invokes cosnstructor2
c2=complex(1.6, 2.7);
c3=c1+c2;
cout<< "c1 = " ;
c1.display();
cout<<"c2 = ";
c2.display();
cout<<"c3 = ";
c3.display();
return 0;
}
```

Here is the output of the above program:

```
c1= 2.5 + j3.5
c2 = 1.6 + j2.7
c3 = 4.1 + j6.2
```

Let us look at the following function:

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

The features of the above function are as follows:

- It receives only one complex type argument explicitly.
- It returns a complex type value.
- It is a member function of complex.

You can observe that the function adds two complex numbers and returns a complex number but it receives only one value as an argument. Now the question is from where does the second value come? Let us now see the following statement which invokes the operator+() function :

```
c3=c1+c2;
```

As you know that a member function can be invoked only by an object of that class. So, here the object c1 invokes the function and c2 is passed as an argument to the function. The above statement is equivalent to:

c3= c1.operator+ (c2) //usual function call syntax

Hence in the operator+ () function, the data members of c1 are accessed directly and the data members of c2 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 c1.temp. x is the real part of temp that has been created to store the result of addition of c1 and c2. The function returns the complex temp to be assigned to c3. The figure 6.1 shows implementation of the above program.

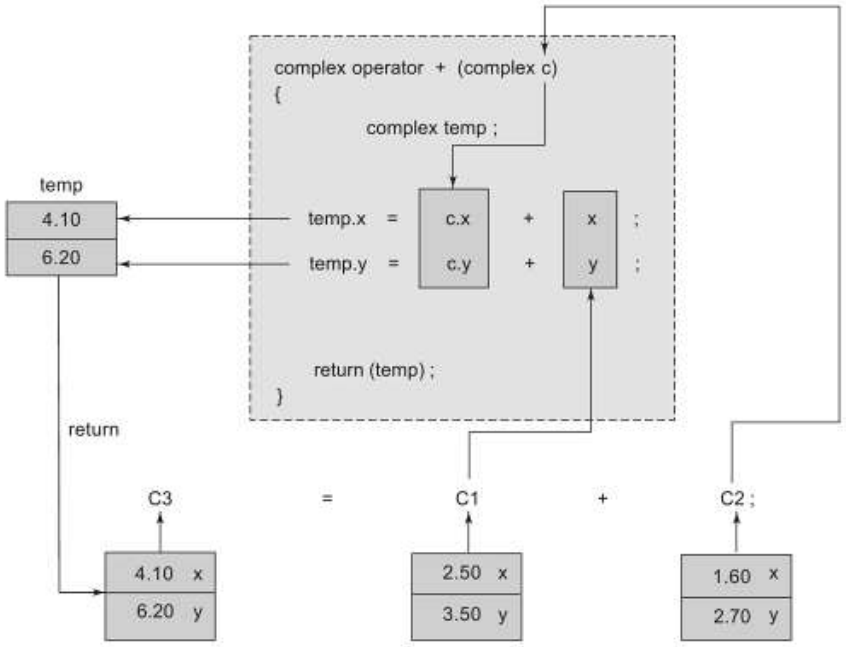

Figure 6.1: Implementation of overloaded + operator

For overloading a binary operator, instead of member functions friend functions can also be used. The difference is that a friend function requires two arguments to be explicitly passed to it whereas a member function requires only one argument.

We can modify the complex number function discussed above using a friend function as shown below:

1. Replace the member function declaration by the friend function declaration like this:
   friend complex operator+ (complex, complex);

2. Redefine the operator function as follows:

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

In this case the statement
c3 = c1 + c2; is equivalent to
c3 = operator+(c1, c2);

In most of the cases same result is generated either by using member functions or by using friend functions. Then the question arises that why an alternative is made available? The reason is that in some situations we prefer using friend function to member function. For example let us consider a situation where it is required to use two different types of operands for binary operator, one is an object and other one is a built in data type as shown below:
A = B + 2; (or A = B * 2);

Where A and B are the objects of same class. This is true if you use a member function but the following statement will not work:
A = 2 + B; (or A = 2 * B);

The reason is that the left hand operator which is responsible for invoking the member function should be an object of the same class. However, friend function allows both the approaches.

It may be recalled that an object need not be used to invoke friend function but can be passed as an argument. Thus, 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.

A program illustrating the use of friend function is shown below:

```
# include<iostream.h>
class distance
{ private:
int feet, inches;
public:
distance()
{feet=0; inches=0;}
distance(int f)
{feet=f; inches=0;}
distance(int f, int i)
{feet=f;inches=i;}
void display()
{cout<<feet <<" "<<inches<<endl;}
friend distance operator + (distance, distance);
};
distance operator + (distance d1, distance d2)
{ int f = d1.feet+d2.feet;
int i = d1.inches+d2.inches;
return distance(f,i);}
void main()
{
distance d1(2,5), d2;
d2=10+d1;
d2.display();
}
```

Let us now study how the relational operators can be overloaded. In this situation also there can be either one argument or two argument overloading (using friend function). But the return value should be integer (0 or 1) which indicates true or false.

The following program implements the program for relational operator overloading (<) for the distance class.

```
# include<iostream.h>
class distance
{ private:
int feet, inches;
public:
distance()
{feet=0; inches=0;}
distance(int f)
{feet=f; inches=0;}
distance(int f, int i)
{feet=f;inches=i;}
void display();
void getdata();
int operator < (distance d1)
{ if (feet<d1.feet)
return 1;
else if (feet==d1.feet) && (inches<d1.inches)
return 1;
else
return 0;
}
};
void distance :: display()
{cout<<feet <<" "<<inches<<endl;}
void distance :: getdata()
{ cout<<”Enter distance in feet and inches”;
cin>>feet >>inches;}
void main()
{
distance d1,d2;
d1.getdata();
d2.getdata();
if (d1<d2)
cout<<d1.display() << “is smaller”;
else
cout<<d2.display()<< “ is smaller”;
}
```

As you can see in the above program the object d1 is responsible for invoking the operator < whereas object d2 is passed as an argument. The value returned by the overloaded operator function is either 1 or 0 which indicates true or false respectively. Certain compilers support boolean datatype which can be alternatively used instead of integer. The display and getdata member functions are implemented in a different way in the above program. You can observe that the member functions are declared inside the class but defined outside the class. The class name with the function name separated by scope resolution operator (::) is used to specify that the member function belongs to the distance class.

Self Assessment Questions

7. Member functions of a class can be defined outside the class using ________ operator along with the class name.
8. In most of the cases same result is generated either by using member function or by using friend function. (True/False).
9. Overloaded relational binary operators should return _________.
10. Left operand calls the operator in case of binary operator overloading. (True/False).

6.5 Type Conversions

An expression can consist of constants and variables of same or different data types. When evaluating an expression which contains a combination of mixed data types, different variables are converted to the same, to avoid compatibility issues. This is achieved by the process of type conversion. Type conversion is defined as the process of converting one predefined data type to another. An assignment operator performs the automatic type conversion. The data that is at the right of the assignment operator is converted to the same data type, which is to the left of assignment operator. For example, consider the following statement:

```
int x;
float y= 4.14532
x=y;
```

In the above example, the compiler converts the value of y to integer before assigning it to the x. The type conversions are implicit for the built in data types. But for the user defined data types, implicit conversions are not supported by the compiler. So, for performing type conversion in user defined data types you have to define the conversion rules.

Three situations may arise in data conversion between incompatible types.

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

We will discuss all the three situations in detail.

1. Basic to class type

The process to convert basic type to class type is easy to perform. Let us consider the following constructor:

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

You can see in the above example that the constructor builds a string type object from a char* type variable x. The variables length and y are data members of string class. Once this constructor is defined in the string class it can be used for conversion from char* type to string type. Let us see another example shown below:

```
string str1, str2;
char* s1= “ABC”;
char* s2= “PQR”;
str1= string( s1);
str2= s2;
```

The statement str1=string(s1) converts s1 from char* type to string type and then assigns the string type value to the object str1. The statement str2=s2 performs the same function by invoking the function implicitly.

2. Class to basic type

The overloaded *casting* operator in C++ allows us to convert class type data to basic types.

The overloaded casting operator has the general form:

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

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

Let us see the following conversion function:

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

By this function the vector is converted to the corresponding scalar magnitude. The magnitude of the vector is calculated by the square root of the sum of the squares of its components. You can use operator double as follows:

```
double length = double(v1);
Or double length = v1;
```

Where v1 is a vector of object type. Both the statements have similar effect. The casting operator is called by the compiler when it comes across the statement that requires the conversion of a class type to a basic type.

The casting operator should meet the following condition:

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

3. One class to another class type

You have studied the data conversion techniques from a basic to class type and a class to basic type. In this section, you are going to study the process to convert one class type data to another class type.

Consider the example:

objA = objB; //objects of different types

Here objA is an object of class A and objB is an object of class B. The class B type data is converted to class A type data and then the converted value is assigned to objA. B is the source class and A is the destination class as conversion takes place from class B to class A.

The conversion between objects of different classes can be performed by either a conversion function or by a constructor. To decide which form is to be used depends upon where you want the type-conversion function to be located i.e. either in the source or destination class. You are already aware that the casting operator function i.e. operator typename() converts a class object of which it is a *member* to *typename.* The typename can be a buit-in type or a user-defined type. Typename indicates the destination class in case of conversions between objects. Therefore, when a class needs to be converted, we can use a casting operator function (i.e. source class). The conversion takes place in the source class and the result is stored in the destination class object.

Now let us consider a single-argument constructor function, which serves as an instruction for converting the argument's type to the class type of which it is a member. This shows that the argument belongs to the destination class and is passed to the destination class for conversion. Then it becomes necessary to place the conversion constructor in the destination class. These two approaches are explained in figure 6.2.

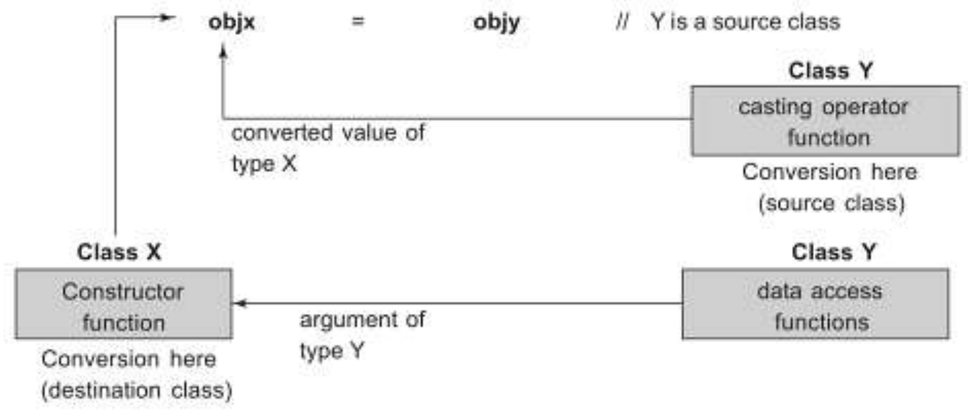
objx = objy // Y is a source class
Class Y
casting operator function
Conversion here (source class)
converted value of type X
Class X
Constructor function
Conversion here (destination class)
argument of type Y
Class Y
data access functions

Figure 6.2: Conversion between objects

Self Assessment Questions

11. ___________ is defined as the process of converting one predefined data type to another.
12. The overloaded _______ operator in C++ allows us to convert class type data to basic types.

6.6 Summary

- Operator overloading allows defining new meaning for normal C++ operators and specifies how it can be used with the user defined classes.
- Overloading should be used only to imply default meanings to avoid confusion in its usage.
- Not all operators can be overloaded. C++ enables the programmer to overload most operators to be sensitive to the context in which they are used.
- The compiler generates the appropriate code based on the operator's use. Operator overloading contributes to C++'s extensibility.
- Operator overloading provides the same concise expressions for user-defined types that C++ provides with its rich collection of operators that work on built-in types.
- The precedence and associativity of an operator cannot be changed by overloading.

- Type conversion is defined as the process of converting one predefined data type to another.
- Three situations may arise in data conversion between incompatible types.
- These are - conversion from basic to class type, conversion from class type to basic type and conversion from one class type to another class type.

6.7 Terminal Questions

1. Create a class string that stores a string value. Overload ++ operator which converts the characters of the string to uppercase (toupper library function of ctype.h can be used).
2. Overload += operator for the string class which should allow statements like s1+=s2. This should concatenate strings s1 and s2 and store the result in s1.
3. Illustrate, with the help of an example, the procedure to overload a unary operator.
4. Explain, with the help pf an example, the process to overload a binary operator.
5. Define type conversion and explain different types of type conversions.

6.8 Answers

Self Assessment Questions

1. True
2. True
3. operator
4. No
5. Same as class datatype
6. False
7. scope resolution operator
8. True
9. Integer or Boolean values
10. True
11. Type Conversion
12. Castings

Terminal Questions

1. //string.cpp

```
# include<iostream.h>
# include<ctype.h>
# include<string.h>
# include<conio.h>
class string
{ char str[25];
public:
string()
{ strcpy(str, "");}
string(char ch[])
{ strcpy(str, ch);}
void display()
{ cout<<str;}
string operator ++()
{string temp;
int i;
for(i=0;str[i]!='\0';i++)
temp.str[i]=toupper(str[i]);s
temp.str[i]='\0';
return temp;
}
};
void main()
{ clrscr();
string s1="hello", s2;
s2=s1++;
s2.display();
getch();
}
```

2. //stringar.cpp
```
# include<iostream.h>
# include<ctype.h>
# include<string.h>
# include<conio.h>
class string
{ char str[25];
public:
string()
{ strcpy(str, "");}
string(char ch[])
{ strcpy(str, ch);}
void display()
{ cout<<str;}
void operator +=(string s2)
{int i,l,j;
l=strlen(str);
for(i=l,j=0; s2.str[j]!='\0'; i++,j++)
str[i]=s2.str[j];
str[i]='\0';
}
};
void main()
{ clrscr();
string s1="hello", s2="world";
s1+=s2;
s1.display();
getch();
}
```
3. In case of overloading of unary operators, the calling operand can be either left or right of the operator as in case of increment and decrement operators. While defining the operator functionality for the class, the keyword operator is used. Refer section 6.3 for more details.

4. In binary operator overloading, one operand is the object of the class invoking the function is passed implicitly to the member operator function. And the other operand is passed as an argument, which can be passed either by value or by reference. Refer section 6.4 for more details.
5. Type conversion is defined as the process of converting one predefined data type to another. Three situations that arise in data conversion between incompatible types are - conversion from basic to class type, conversion from class type to basic type and conversion from one class type to another class type. Refer section 6.5 for more details.

References:

- Object Oriented Programming with C++ - Sixth Edition, by E Balagurusamy. Tata McGraw-Hill Education.
- Object-Oriented Programming using C++, by Satchidananda Dehuri, Alok Kumar Jagadev, Amiya Kumar Rath. PHI Learning Pvt. Ltd.