Operator Overloading in C++

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

A simple and complete example

```
#include<iostream.h>
using namespace std;

class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i =0)  {real = r;   imag = i;}

   // This is automatically called when '+' is used with
   // between two Complex objects
   Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
   }
   void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example call to "operator+"
   c3.print();
}
```

Output:

```
12 + i9
```

What is the difference between operator functions and normal functions?

Operator functions are same as normal functions. The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.

Following is an example of global operator function.

```
#include<iostream.h>
using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}
    void print() { cout << real << " + i" << imag << endl; }

// The global operator function is made friend of this class so
// that it can access private members
friend Complex operator + (Complex const &, Complex const &);
};

Complex operator + (Complex const &c1, Complex const &c2)
{
     return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
    return 0;
}
```

Can we overload all operators?

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

. (dot)

::

?:

sizeof

Why can't . (dot), ::, ?: and sizeof be overloaded?

Important points about operator overloading

1) For operator overloading to work, at leas one of the operands must be a user defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor).

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

```
#include <iostream.h>
using namespace std;
class Fraction
{
   int num, den;
public:
   Fraction(int n,  int d) { num = n; den = d; }

   // conversion operator: return float value of fraction
   operator float() const {
      return float(num) / float(den);
   }
};

int main() {
   Fraction f(2, 5);
   float val = f;
   cout << val;
   return 0;
}
```

Output:

```
0.4
```

Overloaded conversion operators must be a member method. Other operators can either be member method or global method.

4) Any constructor that can be called with a single argument works as a conversion constructor, means it can also be used for implicit conversion to the class being constructed.

```
#include<iostream.h>
class Point
{
private:
   int x, y;
public:
   Point(int i = 0, int j = 0) {
      x = i;   y = j;
   }
   void print() {
      cout << endl << " x = " << x << ", y = " << y;
   }
};
```

```
int main() {
   Point t(20, 20);
   t.print();
   t = 30;   // Member x of t becomes 30
   t.print();
   return 0;
}
```

Output:

```
 x = 20, y = 20

 x = 30, y = 0
```

We will soon be discussing overloading of some important operators like new, delete, comma, function call, arrow, etc.

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
// body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

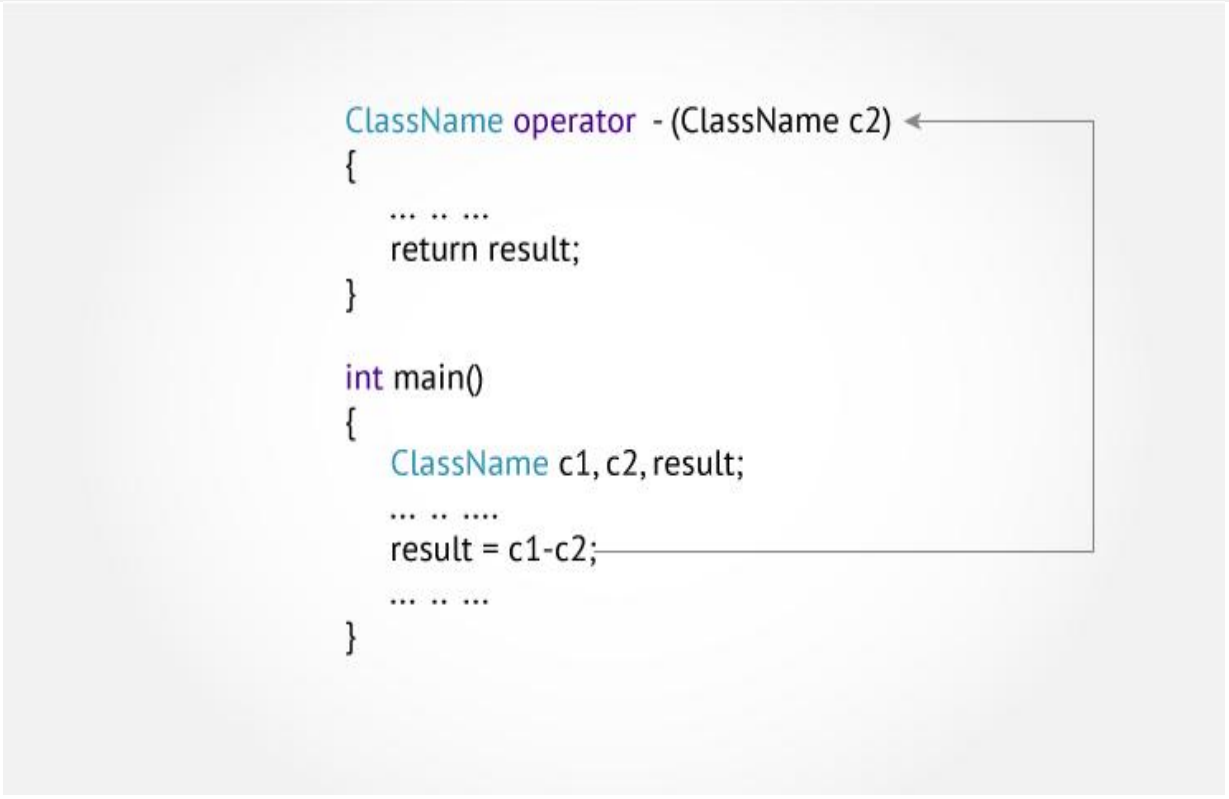
// program to overload the unary operator ++.

```
#include <iostream.h>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()        {
       num = num+2;
     }
     void Print() {
        cout<<"The Count is: "<<num;
     }
};
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Output:

```
The Count is: 10
```

C++ Operator Overloading



```
ClassName operator  - (ClassName c2)
{
    ... .. ...
    return result;
}

int main()
{
    ClassName c1, c2, result;
    ... .. ....
    result = c1-c2;
    ... .. ...
}
```

The meaning of an operator is always same for variable of basic types like: int, float, double etc. For example: To add two integers, + operator is used.

However, for user-defined types (like: objects), you can redefine the way operator works. For example:

If there are two objects of a class that contains string as its data members. You can redefine the meaning of + operator and use it to concatenate those strings.

This feature in C++ programming that allows programmer to redefine the meaning of an operator (when they operate on class objects) is known as operator overloading.

Why is operator overloading used?

You can write any C++ program without the knowledge of operator overloading. However, operator operating are profoundly used by programmers to make program intuitive. For example,

You can replace the code like:

```
calculation = add(multiply(a, b),divide(a, b));
```

to

```
calculation = (a*b)+(a/b);
```

How to overload operators in C++ programming?

To overload an operator, a special operator function is defined inside the class as:

```
class className
{
    ... .. ...
    public
       returnType operator symbol (arguments)
       {
           ... .. ...
       }
    ... .. ...
};
```

- Here, returnType is the return type of the function.
- The returnType of the function is followed by operator keyword.
- Symbol is the operator symbol you want to overload. Like: +, <, -, ++
- You can pass arguments to the operator function in similar way as functions.

Example: Operator overloading in C++ Programming

```
#include <iostream.h.h>
using namespace std;

class Test
{
   private:
      int count;

   public:
       Test(): count(5){}

       void operator ++()
       {
          count = count+1;
       }
       void Display() { cout<<"Count: "<<count; }
};

int main()
{
    Test t;
    // this calls "function void operator ++()" function
    ++t;
    t.Display();
    return 0;
}
```

Output

```
Count: 6
```

This function is called when ++ operator operates on the object of Test class (object t in this case).
In the program,void operator ++ () operator function is defined (inside Test class).
This function increments the value of count by 1 for t object.

Things to remember

1. Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.).
2. Two operators = and & are already overloaded by default in C++. For example: To copy objects of same class, you can directly use = operator. You do not need to create an operator function.
3. Operator overloading cannot change the precedence and associatively of operators. However, if you want to change the order of evaluation, parenthesis should be used.
4. There are 4 operators that cannot be overloaded in C++. They are :: (scope resolution), . (member selection), .* (member selection through pointer to function) and ?: (ternary operator).

Following best practices while using operator overloading

Operator overloading allows you to define the way operator works (the way you want).

In the above example, ++ operator operates on object to increase the value of data member count by 1.

```
void operator ++()
    {
       count = count+1;
    }
```

However, if you use the following code. It decreases the value of count by 100 when ++operator is used.

```
void operator ++()
```

```
    {
       count = count-100;
    }
```

This may be technically correct. But, this code is confusing and, difficult to understand and debug.

It's your job as a programmer to use operator overloading properly and in consistent way.

In the above example, the value of count increases by 1 when ++ operator is used. However, this program is incomplete in sense that you cannot use code like:

```
t1 = ++t
```

It is because the return type of the operator function is void.

Overloadable /Non-Overloadable Operators

Following is the list of operators which can be overloaded −

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

Following is the list of operators, which can not be overloaded −

::	.*	.	?:

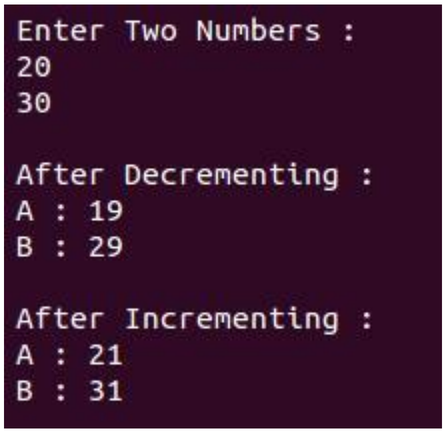
Unary operator overloading - C++ Program

Operator overloading is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. It is used to perform operation on user-defined data type.

Following program is overloading unary operators: increment (++) and decrement (--).

```
#include<iostream.h>
using namespace std;
class IncreDecre
{
        int a, b;
     public:
        void accept()
        {
                cout<<"\n Enter Two Numbers : \n";
                cout<<" ";
                cin>>a;
                cout<<" ";
                cin>>b;
        }
        void operator--() //Overload Unary Decrement
        {
                a--;
                b--;
        }
        void operator++() //Overload Unary Increment
        {
                a++;
                b++;
        }
        void display()
        {
                cout<<"\n A : "<<a;
                cout<<"\n B : "<<b;
        }
};
int main()
{
        IncreDecre id;
        id.accept();
        --id;
        cout<<"\n After Decrementing : ";
        id.display();
        ++id;
        ++id;
        cout<<"\n\n After Incrementing : ";
        id.display();
        return 0;
}
```

Output:



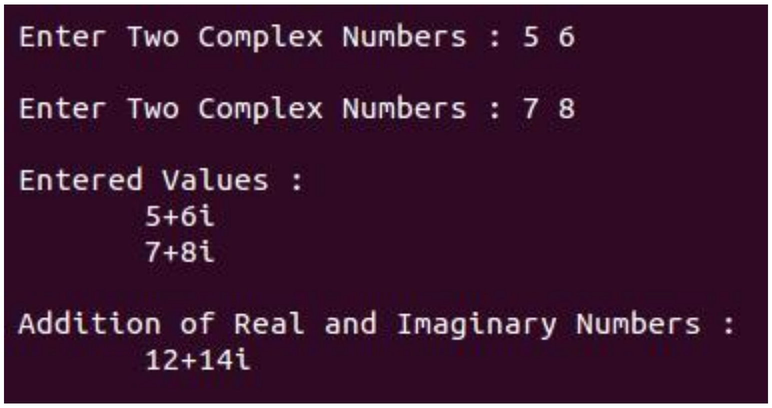
```
Enter Two Numbers :
20
30

After Decrementing :
A : 19
B : 29

After Incrementing :
A : 21
B : 31
```

Binary operator overloading - C++ Program

Following program is overloading binary operator '+' to add two complex numbers.

```
#include<iostream.h>
using namespace std;

class Complex
{
        int num1, num2;
    public:
        void accept()
        {
                cout<<"\n Enter Two Complex Numbers : ";
                cin>>num1>>num2;
        }
        Complex operator+(Complex obj)   //Overloading '+' operator
        {
                Complex c;
                c.num1=num1+obj.num1;
                c.num2=num2+obj.num2;
                return(c);
        }
        void display()
        {
                cout<<num1<<"+"<<num2<<"i"<<"\n";
        }
};
int main()
{
        Complex c1, c2, sum;      //Created Object of Class Complex i.e c1 and c2

        c1.accept();  //Accepting the values
        c2.accept();

        sum = c1+c2;   //Addition of object

        cout<<"\n Entered Values : \n";
        cout<<"\t";
        c1.display();    //Displaying user input values
        cout<<"\t";
        c2.display();

        cout<<"\n Addition of Real and Imaginary Numbers : \n";
        cout<<"\t";
        sum.display();  //Displaying the addition of real and imaginary numbers

        return 0;
}
```

Output:



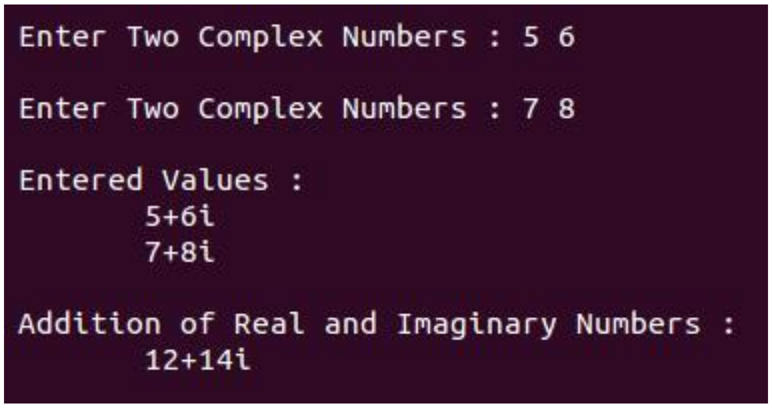
```
Enter Two Complex Numbers : 5 6

Enter Two Complex Numbers : 7 8

Entered Values :
        5+6i
        7+8i

Addition of Real and Imaginary Numbers :
        12+14i
```

Demonstrating operator overloading by using friend function

Following program is demonstrating operator overloading by using friend function.

```
#include<iostream.h>
using namespace std;

class Complex
{
        int num1, num2;
    public:
        void accept()
        {
                cout<<"\n Enter Two Complex Numbers : ";
                cin>>num1>>num2;
        }

        //Overloading '+' operator using Friend function
        friend Complex operator+(Complex c1, Complex c2);

        void display()
        {
                cout<<num1<<"+"<<num2<<"i"<<"\n";
        }
};
Complex operator+(Complex c1, Complex c2)
{
        Complex c;
        c.num1=c1.num1+c2.num1;
        c.num2=c1.num2+c2.num2;
        return(c);
}
int main()
{
        Complex c1,c2, sum;      //Created Object of Class Complex i.e c1 and c2

        c1.accept();  //Accepting the values
        c2.accept();

        sum = c1+c2;   //Addition of object

        cout<<"\n Entered Values : \n";
        cout<<"\t";
        c1.display();    //Displaying user input values
        cout<<"\t";
        c2.display();

        cout<<"\n Addition of Real and Imaginary Numbers : \n";
        cout<<"\t";
        sum.display();  //Displaying the addition of real and imaginary numbers
    return 0;
}
```

Output:



```
Enter Two Complex Numbers : 5 6

Enter Two Complex Numbers : 7 8

Entered Values :
       5+6i
       7+8i

Addition of Real and Imaginary Numbers :
       12+14i
```

Types of Operator Overloading in C++

Operator Overloading:

C++ provides a special function to change the current functionality of some operators within its class which is often called as operator overloading. Operator Overloading is the method by which we can change the function of some specific operators to do some different task.

This can be done by declaring the function, its syntax is,

```
Return_Type classname :: operator op(Argument list)

{

   Function Body

}
```

In the above syntax Return_Type is value type to be returned to another object, operator op is the function where the operator is a keyword and op is the operator to be overloaded.
Operator function must be either non-static (member function) or friend function.

Operator Overloading can be done by using **three approaches**, they are

1. Overloading unary operator.
2. Overloading binary operator.
3. Overloading binary operator using a friend function.

Below are some criteria/rules to define the operator function:

- In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.
- In the case of a friend function, the binary operator should have only two argument and unary should have only one argument.
- All the class member object should be public if operator overloading is implemented.

- Operators that cannot be overloaded are **. .* :: ?:**
- Operator cannot be used to overload when declaring that function as friend function **= () [] ->**.

Refer this, for more rules of Operator Overloading

Note: The arguments in the operator overloading are passed only by reference, it will not work if arguments are passed by value, because a copy of the object is passed to operator (op)() function.

1. **Overloading Unary Operator**: Let us consider to overload (-) unary operator. In unary operator function, no arguments should be passed. It works only with one class objects. It is a overloading of an operator operating on a single operand.

   Example:

   Assume that class Distance takes two member object i.e. feet and inches, create a function by which Distance object should decrement the value of feet and inches by 1 (having single operand of Distance Type).

```
// C++ program to show unary operator overloading
#include <iostream.h>

using namespace std;

class Distance {
public:

   // Member Object
   int feet, inch;

   // Constructor to initialize the object's value
   Distance(int f, int i)
   {
      this->feet = f;
      this->inch = i;
   }

   // Overloading(-) operator to perform decrement
   // operation of Distance object
   void operator-()
   {
      feet--;
      inch--;
      cout << "\nFeet & Inches(Decrement): " << feet << "'" << inch;
   }
};

// Driver Code
int main()
{
   // Declare and Initialize the constructor
   Distance d1(8, 9);
   // Use (-) unary operator by single operand
   -d1;
   return 0;
}
```

Output:

```
Feet & Inches(Decrement): 7'8
```

In the above program, it shows that no argument is passed and no return_type value is returned, because unary operator works on a single operand. (-) operator change the functionality to its member function.

Note: d2 = -d1 will not work, because operator-() does not return any value.

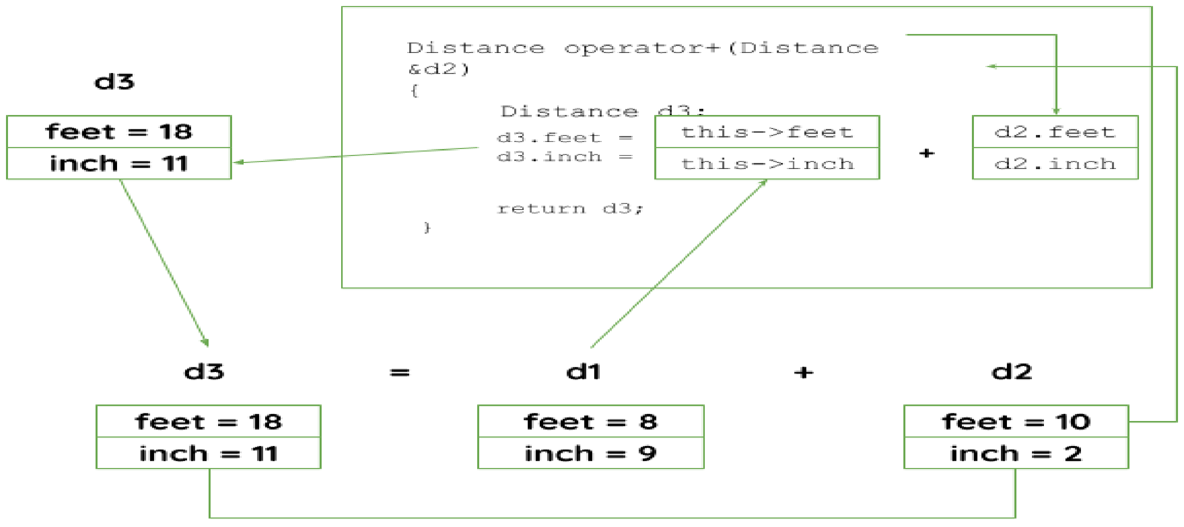
2. **Overloading Binary Operator**: In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands. Let’s take the same example of class Distance, but this time, add two distance objects.

```
// C++ program to show binary operator overloading
#include <iostream.h>

using namespace std;

class Distance {
public:
   // Member Object
   int feet, inch;
   // No Parameter Constructor
   Distance()
   {
      this->feet = 0;
      this->inch = 0;
   }

   // Constructor to initialize the object's value
   // Parametrized Constructor
   Distance(int f, int i)
   {
      this->feet = f;
      this->inch = i;
   }

   // Overloading (+) operator to perform addition of
   // two distance object
   Distance operator+(Distance& d2) // Call by reference
   {
      // Create an object to return
      Distance d3;
```

```
        // Perform addition of feet and inches
        d3.feet = this->feet + d2.feet;
        d3.inch = this->inch + d2.inch;

        // Return the resulting object
        return d3;
    }
};

// Driver Code
int main()
{
    // Declaring and Initializing first object
    Distance d1(8, 9);

    // Declaring and Initializing second object
    Distance d2(10, 2);

    // Declaring third object
    Distance d3;

    // Use overloaded operator
    d3 = d1 + d2;

    // Display the result
    cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
    return 0;
}
```
Output:

```
Total Feet & Inches: 18'11
```

Pictorial View of working of Binary Operator:



Expression Working

3. **Overloading Binary Operator using a Friend function**: In this approach, the operator overloading function must precede with friend keyword, and declare a function class scope. Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator. All the working and implementation would same as binary operator function except this function will be implemented outside of the class scope.
   Let's take the same example using the friend function.

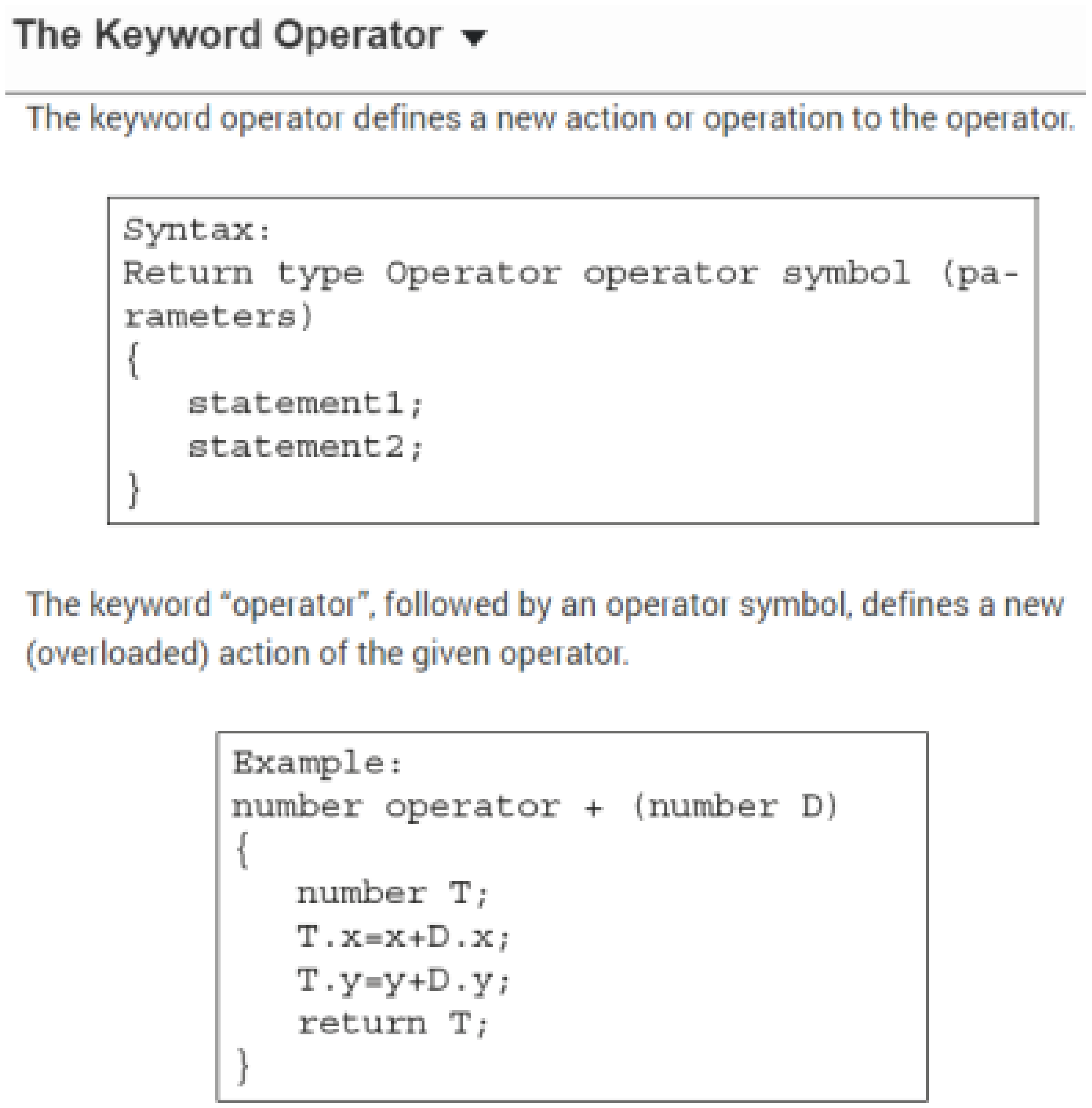
```
// C++ program to show binary operator overloading
#include <iostream.h>

using namespace std;

class Distance {
public:

   // Member Object
   int feet, inch;

   // No Parameter Constructor
   Distance()
   {
      this->feet = 0;
      this->inch = 0;
   }

   // Constructor to initialize the object's value
   // Parametrized Constructor
   Distance(int f, int i)
   {
        this->feet = f;
        this->inch = i;
    }

    // Declaring friend function using friend keyword
    friend Distance operator+(Distance&, Distance&);
};

// Implementing friend function with two parameters
Distance operator+(Distance& d1, Distance& d2) // Call by reference
{
    // Create an object to return
    Distance d3;

    // Perform addition of feet and inches
    d3.feet = d1.feet + d2.feet;
    d3.inch = d1.inch + d2.inch;

    // Return the resulting object
    return d3;
}

// Driver Code
int main()
{
    // Declaring and Initializing first object
    Distance d1(8, 9);

    // Declaring and Initializing second object
    Distance d2(10, 2);

    // Declaring third object
    Distance d3;

    // Use overloaded operator
    d3 = d1 + d2;

    // Display the result
    cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
    return 0;
}
```

Output:

```
Total Feet & Inches: 18'11
```

Here in the above program, operator function is implemented outside of class scope by declaring that function as the friend function.

In these ways, an operator can be overloaded to perform certain tasks by changing the functionality of operators.



The Keyword Operator ▾

The keyword operator defines a new action or operation to the operator.

```
Syntax:
Return type Operator operator symbol (pa-
rameters)
{
   statement1;
   statement2;
}
```

The keyword “operator”, followed by an operator symbol, defines a new (overloaded) action of the given operator.

```
Example:
number operator + (number D)
{
   number T;
   T.x=x+D.x;
   T.y=y+D.y;
   return T;
}
```

Overloaded operators are redefined within a C++ class using the keyword operator followed by an operator symbol. When an operator is overloaded, the produced symbol is called the operator function name. The above declarations provide an extra meaning to the operator. Operator functions should be either member functions or friend functions. A friend function requires one argument for unary operators and two for binary operators, while a member function requires one argument for binary operators and no argument for unary operator. When the member function is called, the calling object is passed implicitly to the function and hence available for member function. While using friend functions, it is essential to pass the objects by value or reference. The prototype of operator functions in classes can be written as follows.

```
(a) void operator ++();
(b) void operator --();
(c) void operator -();
(d) num operator+(num);
(e) friend num operator * (int ,num);
(f) void operator = (num);
```

Operator Return Type ▾

In the last few examples we declared the operator() of void types, that is, it will not return any value. However, it is possible to return value and assign to it other object of the same type. The return value of operator is always of class type, because the operator overloading is only for objects. An operator cannot be overloaded for basic data type. Hence, if the operator returns any value, it will be always of class type. Consider the following program.

10.5 Write a program to return values from operator() function.

```
#include<iostream.h>
#include<conio.h>
class plusplus
{
   private:
   int num;
   public :
   plusplus() { num=0; }
   int getnum() { return num;}
   plusplus operator ++ (int)
   {
   plusplus tmp;
   num=num+1;
   tmp.num=num;
   return tmp;
   }
};
```

```
void main()
{
   clrscr();
   plusplus p1, p2;
   cout<<"\n p1="<<p1.getnum();
   cout<<"\n p2="<<p2.getnum();
   p1=p2++;
   cout<<endl<<" p1="<<p1.getnum();
   cout<<endl<<" p2="<<p2.getnum();
   p1++;
   // p1++=2;
   cout<<endl<<" p1="<<p1.getnum();
   cout<<endl<<" p2="<<p2.getnum();
}
```

OUTPUT

```
p1 = 0
p2 = 0
p1 = 1
p2 = 1
p1 = 2
p2 = 1
```

Explanation: In the above program class plusplus is declared with one private integer num. The class constructor initializes the object with zero. The member function getnum() returns current value of variable num. The operator ++() is overloaded and it can handle as postfix increment of the objects. In case of an increase in the prefix it will flag an error.

The p1 and p2 are objects of the class plusplus. The statement p1=p2++ first increments the value of p2 and then assigns it to the object p1. The values displayed will be one for the objects. The object p1 is increased. This time, the values of object displayed will be two and one.