Operator overloading

- It is one of the many exciting features of C++.
- Important technique that has enhanced the power of extensibility of C++.
- C++ tries to make the user-defined data types behave in much the same way as the built-in types.
- C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types.

Operator overloading

- Addition (+) operator can work on operands of type char, int, float & double.
- However, if s1, s2, s3 are objects of the class string, the we can write the statement,

  s3 = s1 + s2;

- This means C++ has the ability to provide the operators with a special meaning for a data type.
- Mechanism of giving special meaning to an

Operator overloading

- Operator – is a symbol that indicates an operation.
- Overloading – assigning different meanings to an operation, depending upon the context.
- For example: input(>>)/output(<<) operator
  - The built-in definition of the operator << is for shifting of bits.
  - It is also used for displaying the values of various data types.

Operator overloading

- We can overload all C++ operator except the following:
  - Class member access operator (. , .*)
  - Scope resolution operator(::)
  - Size operator (sizeof)
  - Conditional operator(?:)

Defining operator overloading

- The general form of an operator function is:

```
return-type class-name :: operator op (argList)
{
      function body // task defined.
}
```

  - where `return-type` is the type of value returned by the specified operation.
  - `op` is the operator being overloaded.
  - `operator op` is the function name, where operator is a keyword.

Operator overloading

- When an operator is overloaded, the produced symbol called operator function name.
- operator function should be either member function or friend function.
- Friend function requires one argument for unary operator and two for binary operators.
- Member function requires one arguments for binary operators and zero arguments for unary operators.

Operator overloading

Process of overloading involves following steps:

1. Creates the class that defines the data type i.e. to be used in the overloading operation.
2. Declare the operator function operator op() in the public part of the class. It may be either a member function or friend function.
3. Define the operator function to implement the required operations.

Overloading unary operator

- Overloading devoid of explicit argument to an operator function is called as unary operator overloading.
- The operator ++, -- and – are unary operators.
- ++ and -- can be used as prefix or suffix with the function.
- These operators have only single operand.

Overloading Unary Operators (-)

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

class UnaryOp
{
    int x,y,z;
 public:

    UnaryOp()
    {
        x=0;
        y=0;
        z=0;
    }

    UnaryOp(int a,int b,int c)
    {
        x=a;
        y=b;
        z=c;
    }

    void display()
    {
        cout<<"\n\n\t"<<x<<"  "<<y<<"    "<<z;
    }

    // Overloaded minus (-) operator
    void operator- ();
};
```

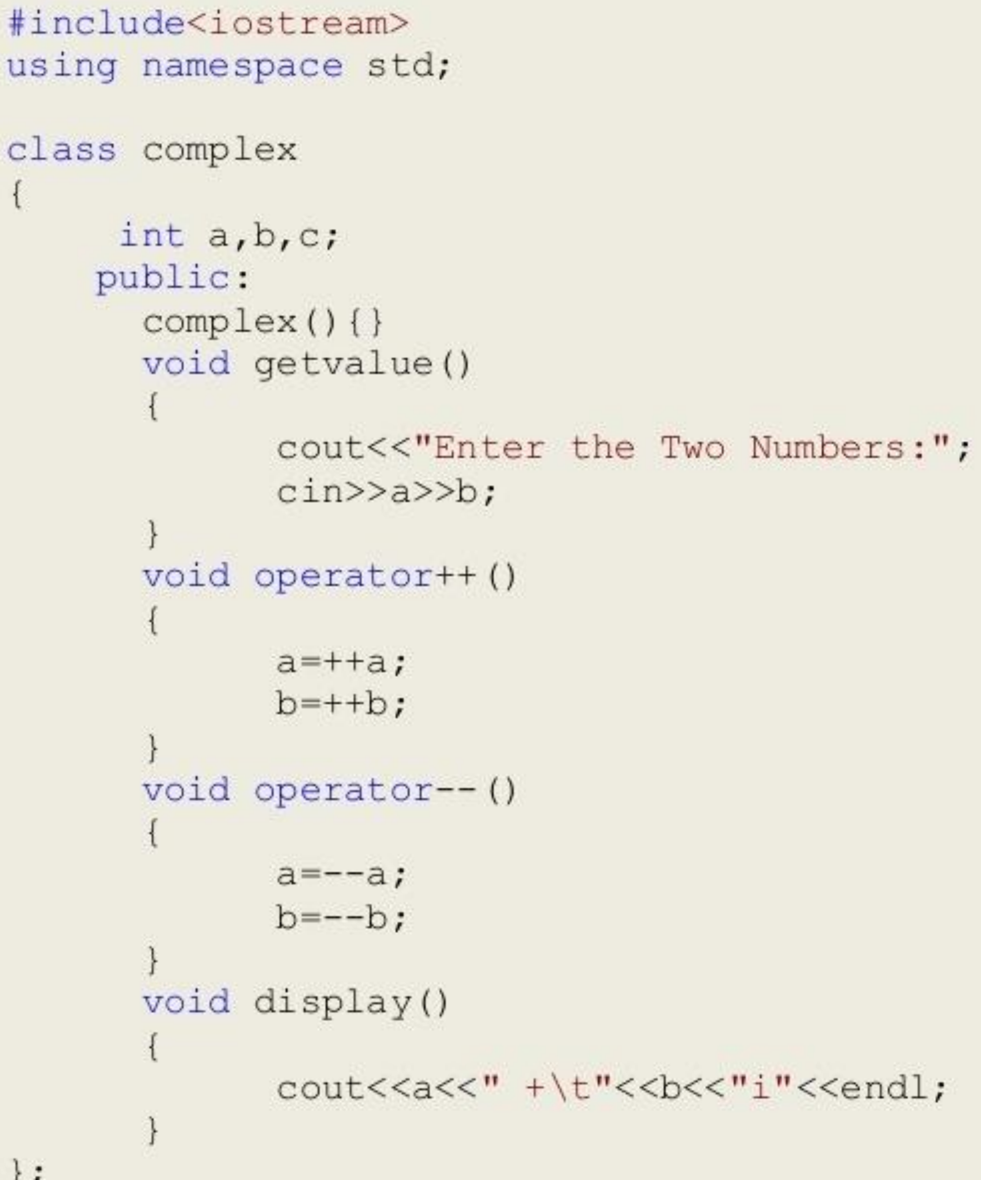
Overloading Unary Operators (-)

```
void UnaryOp ::  operator- ()
{
    x= -x;
    y= -y;
    z= -z;
}

int main()
{
    UnaryOp un(10,-40,70);
    cout<<"\n\nNumbers are :::\n";
    un.display();
    -un;               // call unary minus operator function
    cout<<"\n\nNumbers are after overloaded minus (-) operator :::\n";
    un.display();  // display un
    return 0;
}
```

Output :

```
Numbers are :::
        10  -40    70
Numbers are after overloaded minus (-) operator :::
        -10  40    -70
```

Overloading Unary Operators (++/--)



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

class complex
{
     int a,b,c;
    public:
      complex(){}
      void getvalue()
      {
            cout<<"Enter the Two Numbers:";
            cin>>a>>b;
      }
      void operator++()
      {
            a=++a;
            b=++b;
      }
      void operator--()
      {
            a=--a;
            b=--b;
      }
      void display()
      {
            cout<<a<<" +\t"<<b<<"i"<<endl;
      }
};
```

Overloading Unary Operators (++/--)

```
int main()
{
    complex obj;
    obj.getvalue();
    obj++;
    cout<<"Increment Complex Number\n";
    obj.display();
    obj--;
    cout<<"Decrement Complex Number\n";
    obj.display();
    return 0;
}
```

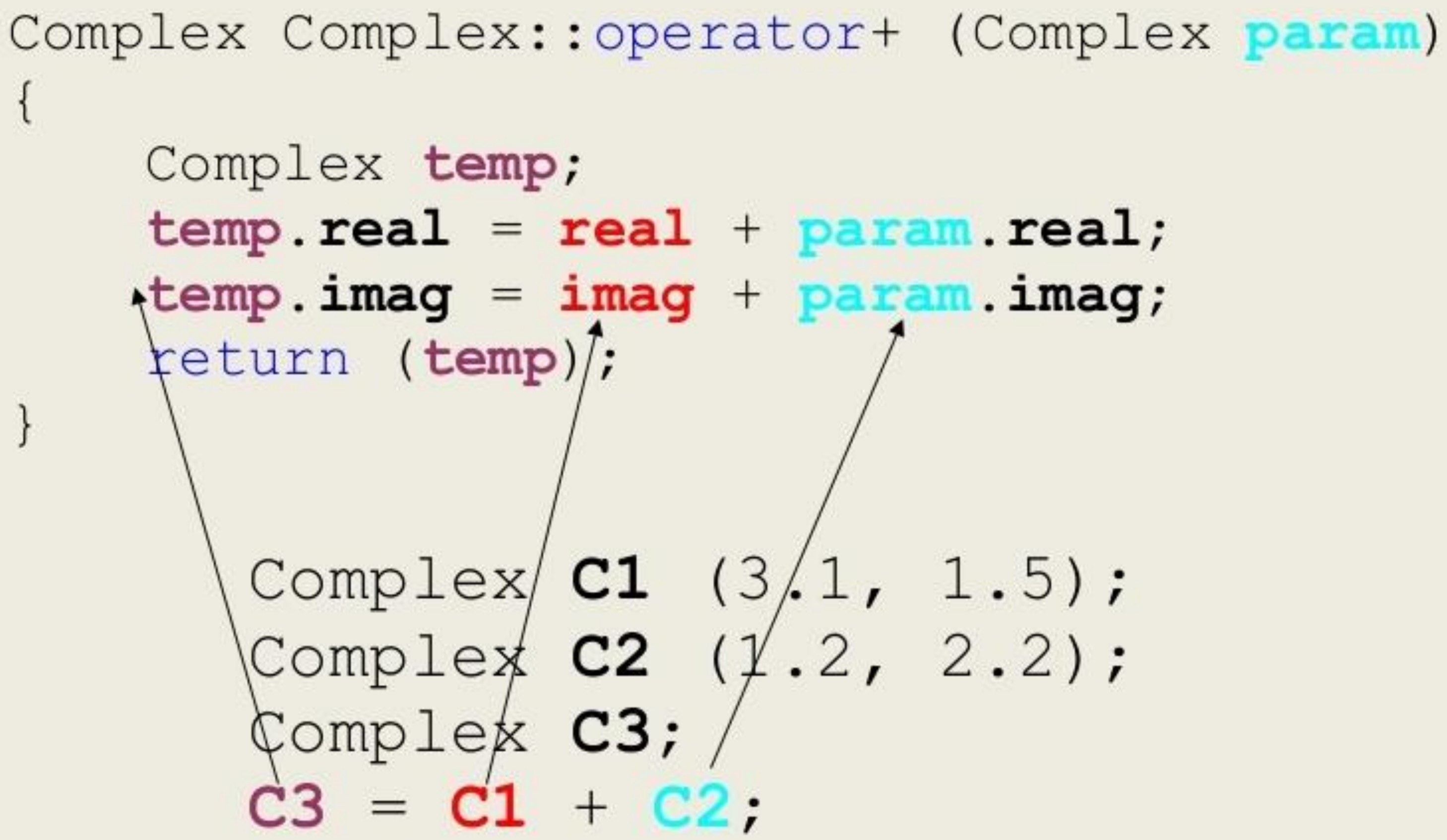
Output:

```
Enter the Two Numbers:
2
3
Increment Complex Number
3  +      4i
Decrement Complex Number
2  +      3i
```

Overloading Binary Operators (+)

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

class Complex
{
        double real;
        double imag;
    public:
        Complex () {}
        Complex (double, double);
        Complex operator + (Complex);
        void print();
};

Complex::Complex (double r, double i)
{
    real = r;
    imag = i;
}

Complex Complex::operator+ (Complex param)
{
    Complex temp;
    temp.real = real + param.real;
    temp.imag = imag + param.imag;
    return (temp);
}
```

Overloading Binary Operators (+)



```
Complex Complex::operator+ (Complex param)
{
    Complex temp;
    temp.real = real + param.real;
    temp.imag = imag + param.imag;
    return (temp);
}


      Complex C1 (3.1, 1.5);
      Complex C2 (1.2, 2.2);
      Complex C3;
      C3 = C1 + C2;
```

Two objects c1 and c2 are two passed as an argument. c1 is treated as first operand and c2 is treated as second operand of the + operator.

Programming Exercise:

Write a program to find out factorial of given number using ‘*’ function.

Overloading Binary Operators (+)

```
void Complex::print()
{
    cout << real << " + i" << imag << endl;
}

int main ()
{
    Complex c1 (3.1, 1.5);
    Complex c2 (1.2, 2.2);
    Complex c3;

    c3 = c1 + c2; //use overloaded + operator
      //c3 = c1.operator+(c2);
    c1.print();
    c2.print();
    c3.print();
    return 0;
}
```

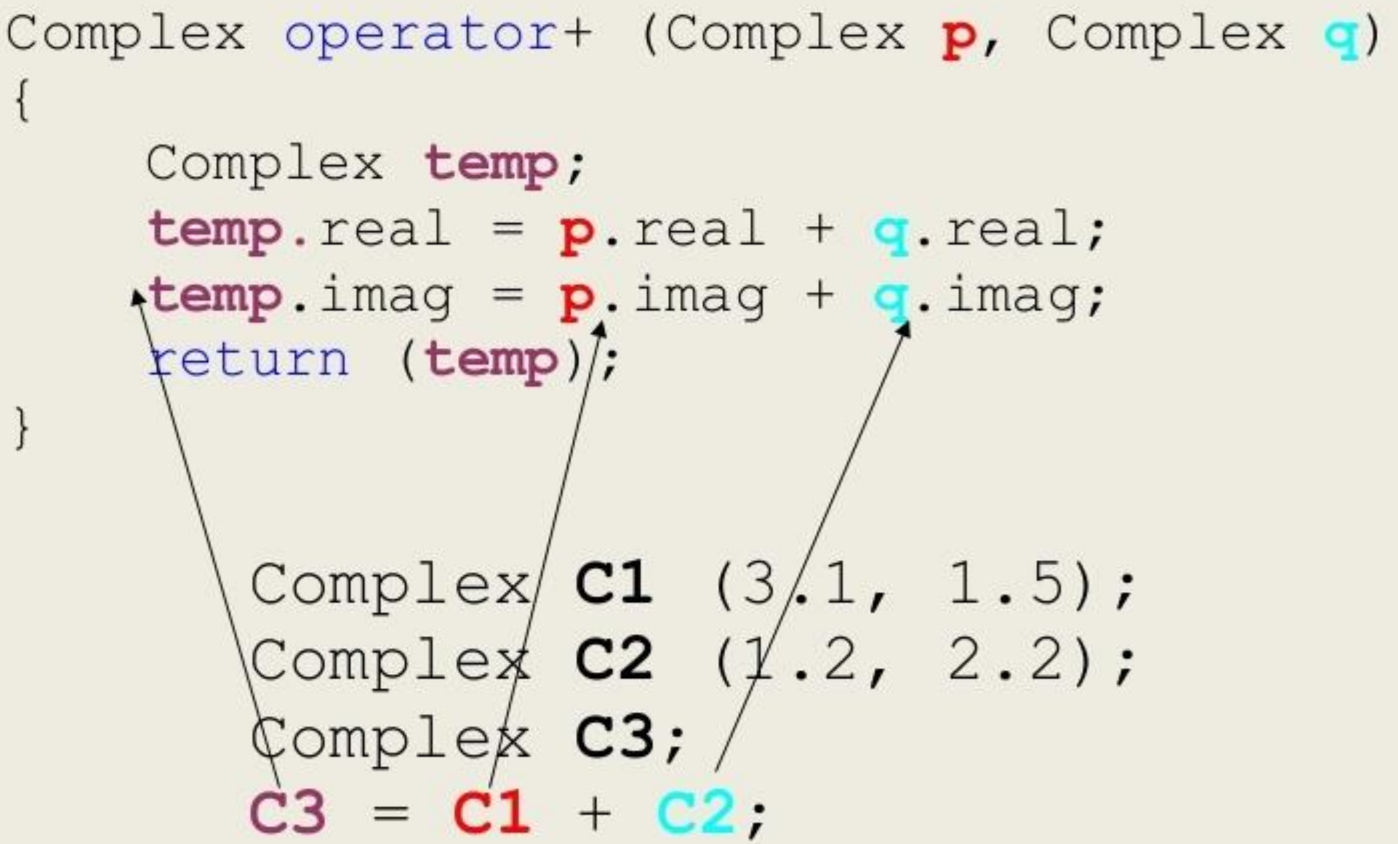
Output :

```
3.1 + i 1.5
1.2 + i 2.2
4.3 + i 3.7
```

Overloading Binary Operators (+) using friend function

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

class Complex
{
        double real;
        double imag;
    public:
        Complex () {}
        Complex (double, double);
        friend Complex operator + (Complex, Complex);
        void print();
};

Complex::Complex (double r, double i)
{
    real = r;
    imag = i;
}

Complex operator+ (Complex p, Complex q)
{
    Complex temp;
    temp.real = p.real + q.real;
    temp.imag = p.imag + q.imag;
    return (temp);
}
```

Overloading Binary Operators (+) using friend function



```
Complex operator+ (Complex p, Complex q)
{
    Complex temp;
    temp.real = p.real + q.real;
    temp.imag = p.imag + q.imag;
    return (temp);
}
```

```
        Complex C1 (3.1, 1.5);
        Complex C2 (1.2, 2.2);
        Complex C3;
        C3 = C1 + C2;
```

Two objects c1 and c2 are two passed as an argument. c1 is treated as first operand and c2 is treated as second

Overloading Binary Operators (+) using friend function

```
void Complex::print()
{
    cout << real << " + i" << imag << endl;
}

int main ()
{
    Complex c1 (3.1, 1.5);
    Complex c2 (1.2, 2.2);
    Complex c3;

    c3 = c1 + c2; //use overloaded + operator
    //c3 = operator+(c1, c2);
    c1.print();
    c2.print();
    c3.print();
    return 0;
}
```

Output :

```
3.1 + i 1.5
1.2 + i 2.2
4.3 + i 3.7
```

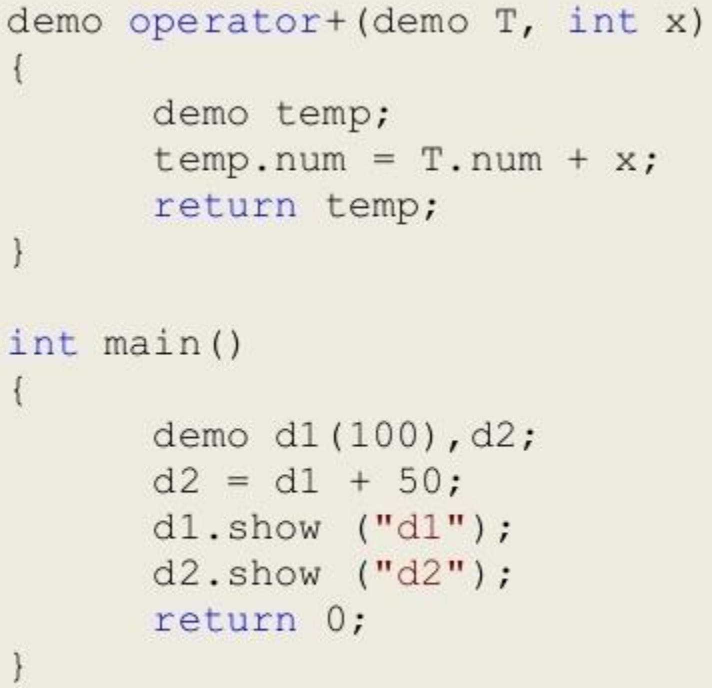
Why to use friend function?

- Consider a situation where we need to use two different types of operands for binary operator.
- One an object and another a built-in –type data.
- d2 = d1 + 50;

Why to use friend function?

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

class demo
{
        int num;
 public:
        demo()
        {
                num = 0;
        }
        demo(int x)
        {
                num = x;
        }
        friend demo operator+(demo, int);
        void show(char *s)
        {
                cout << "num of object "<< s << "=" << num <<endl;
        }
};
```

Why to use friend function?



```
demo operator+(demo T, int x)
{
      demo temp;
      temp.num = T.num + x;
      return temp;
}

int main()
{
      demo d1(100),d2;
      d2 = d1 + 50;
      d1.show ("d1");
      d2.show ("d2");
      return 0;
}
```

Output :

```
num of object d1=100
num of object d2=150
```

Overloading Input/output operator

- C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<.
- Overloaded to perform input/output for user defined data types.
- Left Operand will be of types ostream & and istream &.
- Function overloading this operator must be a Non-Member function because left operand is not an Object of the class.
- It must be a friend function to access private data members.

Overloading Input/output operator

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

class time
{
      int hr,min,sec;
 public:
      time()
      {
            hr=0, min=0; sec=0;
      }

      time(int h,int m, int s)
      {
            hr=h, min=m; sec=s;
      }
      friend ostream& operator << (ostream &out, time &tm);
      //overloading '<<' operator
};
```

Overloading Input/output operator

```
ostream& operator << (ostream &out, time &tm)   //operator function
{
      out << "Time is " << tm.hr << "hour : " << tm.min << "min : "
          << tm.sec << "sec";
      return out;
}

int main()
{
      time tm(3,15,45);
      cout << tm;
      return 0;
}
```

Output:

```
Time is 3 hour : 15 min : 45 sec
```

Overloading Input/output operator

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

class dist
{
      int feet;
      int inch;
  public:
      dist()
      {
            feet = 0;
            inch = 0;
      }
      dist(int a, int b)
      {
            feet = a;
            inch = b;
      }
      friend ostream& operator <<(ostream &out, dist &d);
      friend istream& operator >>(istream &in, dist &d);
};
```

Overloading Input/output operator

```
ostream& operator <<(ostream &out, dist &d)
{
      out <<"Feet::" << d.feet << " Inch::" << d.inch <<endl;
      return out;
}
istream& operator >>(istream &in, dist &d)
{
      in >> d.feet >> d.inch;
      return in;
}
int main()
{
      dist d1(11, 10), d2(5, 11), d3;
      cout <<"Enter the values of object:"<<endl;
      cin >> d3;

      cout <<"First Distance :"<<d1<<endl;
      cout <<"Second Distance :"<<d2<<endl;
      cout <<"Third Distance :"<<d3<<endl;
      return 0;
}
```

Output ::

```
Enter the values of object:
11
```

Overloading Assignment(=) operator

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

class dist
{
      int feet;
      int inch;
  public:
      dist()
      {
            feet = 0;
            inch = 0;
      }
      dist(int a, int b)
      {
            feet = a;
            inch = b;
      }
      void operator = (dist &d)
      {
            feet = d.feet;
            inch = d.inch;
      }
      void display ()
      {
            cout << "Feet: " << feet << " Inch: " << inch << endl;
      }
};
```

Overloading Assignment(=) operator

```
int main()
{
      dist d1(11, 10), d2(5, 11);
      cout <<"First Distance :"<< endl;
      d1.display ();
      cout <<"Second Distance :"<< endl;
      d2.display ();
      //use of asssignment operator
      d1 = d2;
      cout <<"First Distance :"<< endl;
      d1.display ();
      return 0;
}
```

Output::

```
First Distance :
Feet: 11 Inch: 10
Second Distance :
Feet: 5 Inch: 11
First Distance :
Feet: 5 Inch: 11
```

Overloading Arithmetic assignment (+=) operator

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

class dist
{
      int feet;
      int inch;
  public:
      dist()
      {
            feet = 0;
            inch = 0;
      }
      dist(int a, int b)
      {
            feet = a;
            inch = b;
      }
      void display ()
      {
            cout << "Feet: " << feet << " Inch: " << inch << endl;
      }
      void operator += (dist &d)
      {
            feet += d.feet;
            inch += d.inch;
      }
};
```

Overloading Arithmetic assignment (+=) operator

```
int main()
{
      dist d1(11, 10), d2(5, 11);
      cout <<"First Distance :"<< endl;
      d1.display ();
      cout <<"Second Distance :"<< endl;
      d2.display ();
      d1 += d2;
      cout <<"First Distance :"<< endl;
      d1.display ();
      return 0;
}
```

Output ::

```
First Distance :
Feet: 11 Inch: 10
Second Distance :
Feet: 5 Inch: 11
First Distance :
Feet: 16 Inch: 21
```

Overloading Subscript ([]) operator

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

class demo
{
      int *p;
public:
      demo(int n)
      {
            p = new int [n];
            for(int i = 0; i < n; i++)
                  p[i] = i + 1;
      }
      int operator[](int x)
      {
            return p[x];
      }
};
```

Overloading Subscript ([]) operator

```
int main()
{
      demo d(5);
      for(int i = 0; i < 5; i++)
            cout << d[i]<< " ";
      return 0;
}
```

Output ::

```
1 2 3 4 5
```

Statement d[i] is interpreted internally as d.operator[](x). In each iteration of for loop we call the overloaded operator function [] and pass the value of 'i' which returns the corresponding array elements.

Overloading relational operator

- There are various relational operators supported by c++ language which can be used to compare c++ built-in data types.
- For Example:
  - Equality (==)
  - Less than (<)
  - Less than or equal to (<=)
  - Greater than (>)
  - Greater than or equal to (>=)
  - Inequality (!=)
- We can overload any of these operators, which can be used to compare the objects of a class.

Overloading relational operator

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

class dist
{
      int feet;
      int inch;
  public:
      dist(int a, int b)
      {
            feet = a;
            inch = b;
      }
      void display ()
      {
            cout << "Feet: " << feet << " Inch: " << inch << endl;
      }

      bool operator < (dist d)
      {
         if(feet < d.feet)
         {
            return true;
         }
         if(feet == d.feet && inch < d.inch)
         {
            return true;
         }
         return false;
      }
```

Overloading relational operator

```
int main()
{
      dist d1(11, 10), d2(5, 11);
      cout <<"First Distance :"<< endl;
      d1.display ();
      cout <<"Second Distance :"<< endl;
      d2.display ();

      if (d1 < d2)
            cout << "d1 is less than d2." << endl;
      else
            cout << "d1 is greater than (or equal to) d2." << endl;
      return 0;
}
```

Output::

```
First Distance :
Feet: 11 Inch: 10
Second Distance :
Feet: 5 Inch: 11
d1 is greater than (or equal to) d2.
```

Overloading pointer-to-member (->) operator

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

class test
{
        int num;
public:
        test (int j)
        {
                num = j;
        }
        void display()
        {
                cout << "num is " << num << endl;
        }
        test *operator ->(void)
        {
                return this;
        }
};
```

The ‘this’ pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions. ‘this’ pointer is a constant pointer that

Overloading pointer-to-member (->) operator

```
int main()
{
      test T (5);
      T.display ();     //acessing display() normally
      test *ptr = &T;
      ptr -> display(); //using class pointer
      T -> display();     //using overloaded operator
      return 0;
}
```

Output::

```
num is 5
num is 5
num is 5
```

Rules for overloading operator

- Only existing operators can be overloaded. We cannot create a new operator.
- Overloaded operator should contain one operand of user-defined data type.
  - Overloading operators are only for classes. We cannot overload the operator for built-in data types.
- Overloaded operators have the same syntax as the original operator.
- Operator overloading is applicable within the scope (extent) in which overloading occurs.
- Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

Rules for overloading operator

- Overloading of an operator cannot change the basic idea of an operator.
  - For example A and B are objects. The following statement
  - A+=B;
  - assigns addition of objects A and B to A.
  - Overloaded operator must carry the same task like original operator according to the language.
  - Following statement must perform the same operation like the last statement.
  - A=A+B;
- Overloading of an operator must never change its natural meaning.
  - An overloaded operator + can be used for subtraction of two objects, but this type of code decreases the

Type Conversion

- C++ allows to convert one data type to another e.g. int ›››› float
- For example:

```
int m ;
float x = 3.1419;
m = x;
```

- convert x to an integer before its values is assigned to m. Thus, fractional part is truncated.
- C++ already knows how to convert between built-in data types.
- However, it does not know how to convert any user-defined classes

Type Conversion

There are three possibilities of data conversion as given below:

1. Conversion from basic-data type to user-defined data type.
2. Conversion from class type to basic-data type.
3. Conversion from one class type to another class type.

Type Conversion

Basic to Class data type conversion:

- Conversion from basic to class type is easily carried out.
- It is automatically done by compiler with the help of in-built routines or by typecasting.
- Left-hand operand of = sign is always class type and right-hand operand is always basic type.

Conversion from Basic to class-type:

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

class time
{
      int hrs;
      int min;
   public:
      time()
      {
            hrs = 0;
            min = 0;
      }
      time(int t)
      {
            hrs = t / 60;
            min = t % 60;
      }
      void display ()
      {
            cout << hrs << "::" << min <<endl;
      }
};
```

Conversion from Basic to class-type:

```
int main()
{
        time T;
        int duration = 85;
        T = duration;
        T.display();
        return 0;
}
```

Output ::

```
1::25
```

Type Conversion

Class to basic-data type conversion :

- In this conversion, the programmer explicitly tell the compiler how to perform conversion from class to basic type.
- These instructions are written in a member function.
- Such function is known as overloading of type cast operators.
- Left-hand operand is always Basic type and right-hand operand is always class type.

Type Conversion

Class-type to basic-data type conversion :

- While carrying this conversion, the statement should satisfy the following conditions:
  1. The conversion function should not have any argument.
  2. Do not mention return type.
  3. It should be class member function.

Conversion from Class to Basic-type:

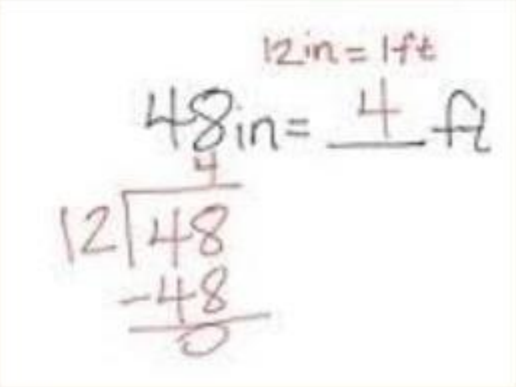
```
#include<iostream>
using namespace std;

class Distance
{
        int length;
public:
        Distance (int n)
        {
                length = n;
        }
        operator int()
        {
                return length;
        }
};
int main()
{
        Distance d(12);
        int len = d;                    // implicit
        int hei = (int) d; // Explicit
        cout << hei;
        return 0;
}
```

We have converted Distance class object into integer type. When the statement int len = d; executes, the compiler searches for a function which can convert an object of Distance class type to int type.

Type Conversion

Conversion from one Class to another Class Type:

- When an object of one class is passed to another class, it is necessary clear-cut instructions to the compiler.
- How to make conversion between these two user defined data types?



Conversion from one class to another class-type:

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

class nInch
{
      int inch;
public:
      nInch (int n)
      {
            inch = n;
      }
      int getInch()
      {
            return inch;
      }
};
```

Conversion from one class to another class-type:

```
class nFeet
{
      int feet;
public:
      nFeet (int n)
      {
            feet = n;
      }
      operator nInch()
      {
            return nInch(feet * 12);
      }
      friend void printInch(nInch m);
};
void printInch(nInch m)
{
      cout << m.getInch ();
}
```

Conversion from one class to another class-type:

```
int main()
{
      int n;
      cout << "Enter feet: " << endl;
      cin >> n;
      nFeet f(n);
      cout << "Inch is : ";
      printInch (f);
      return 0;
}
```

Output:

```
Enter feet:
2
Inch is : 24
```