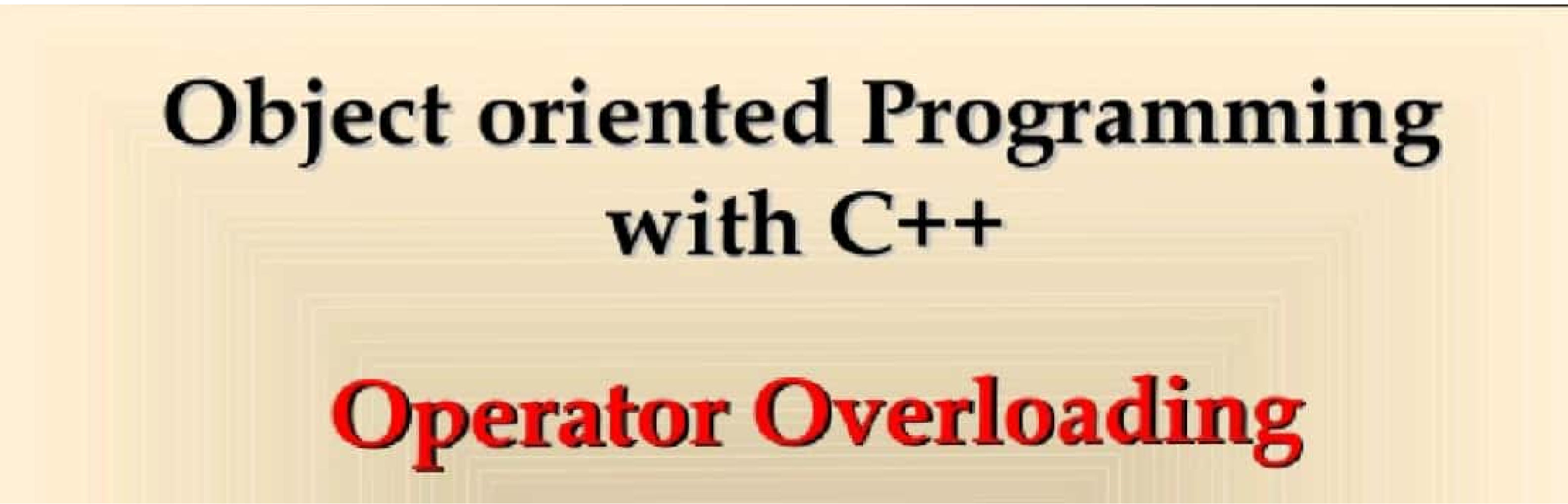
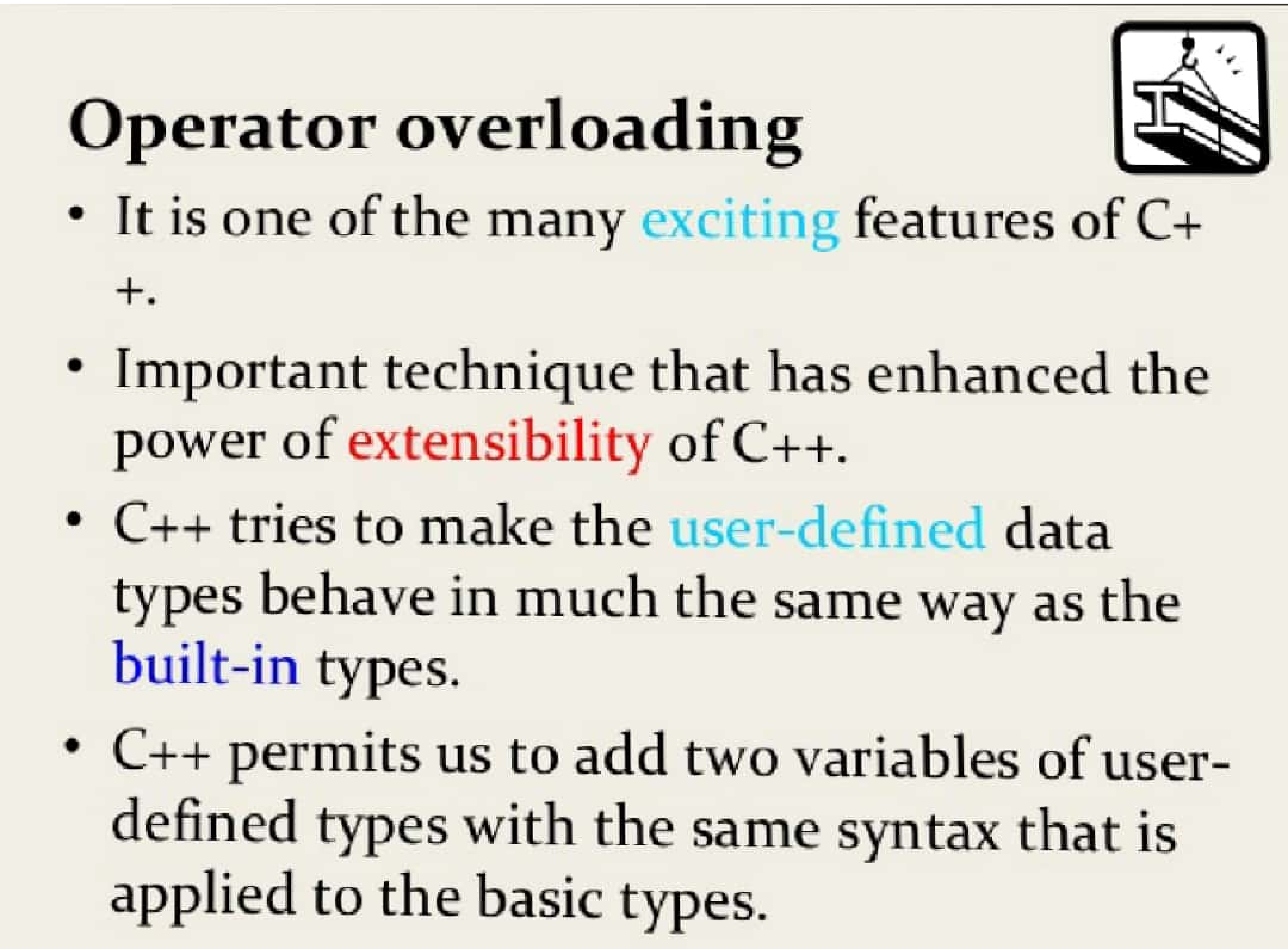


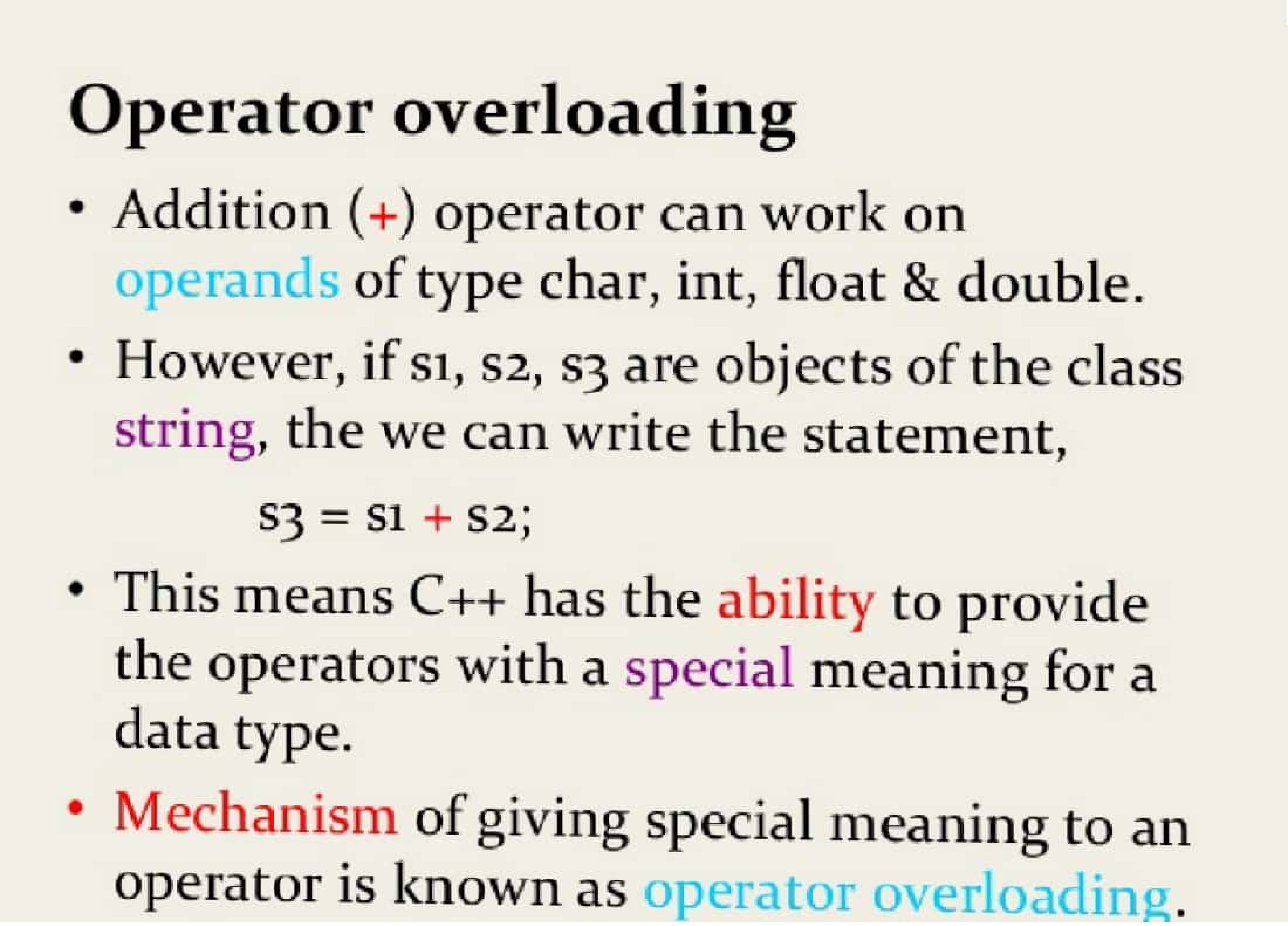
Object oriented Programming with C++

Operator Overloading



Operator overloading

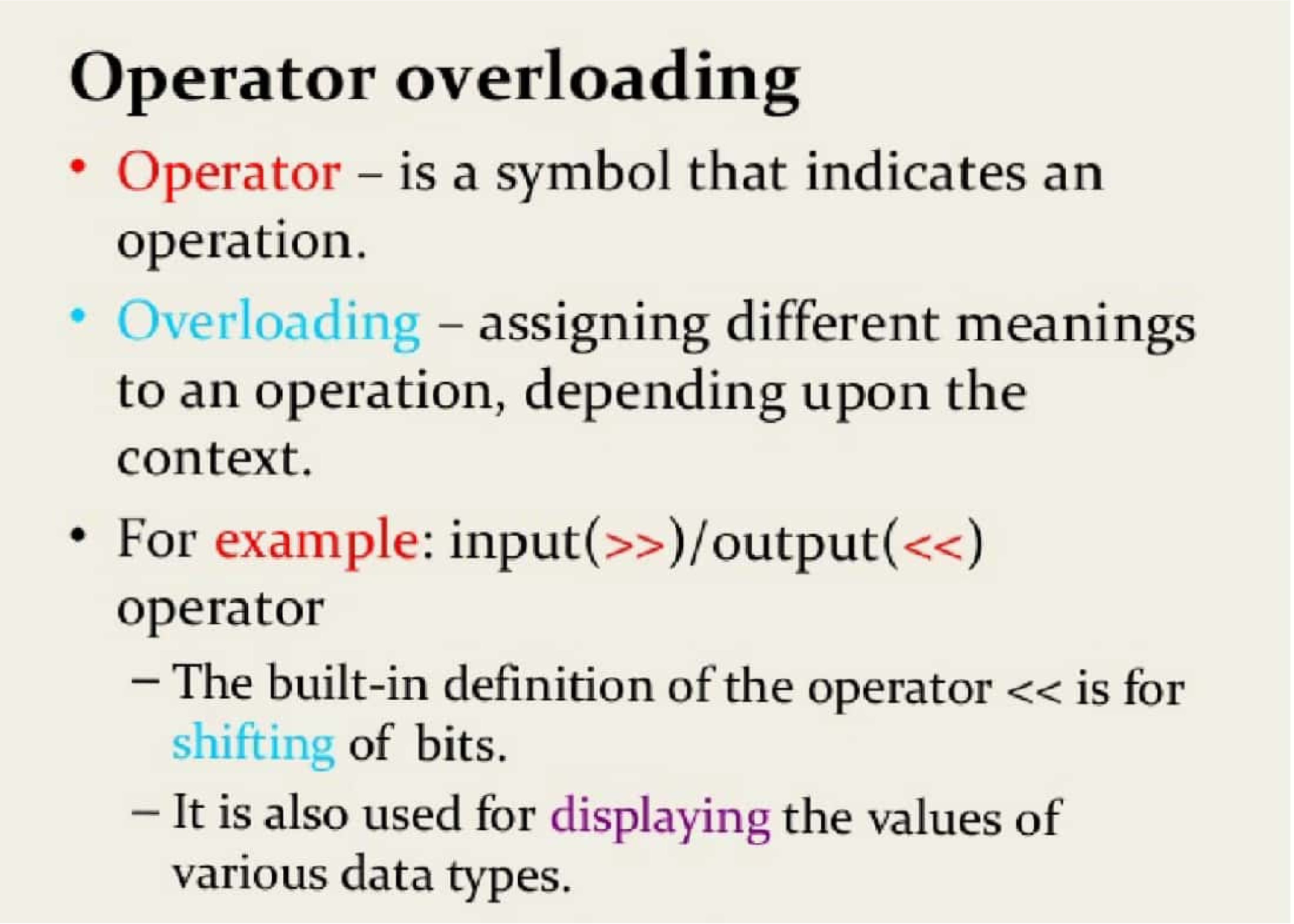
- It is one of the many exciting features of C++.
- Important technique that has enhanced the power of extensibility of C++.
- C++ tries to make the user-defined data types behave in much the same way as the built-in types.
- C++ permits us to add two variables of user-defined types with the same syntax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



Operator overloading

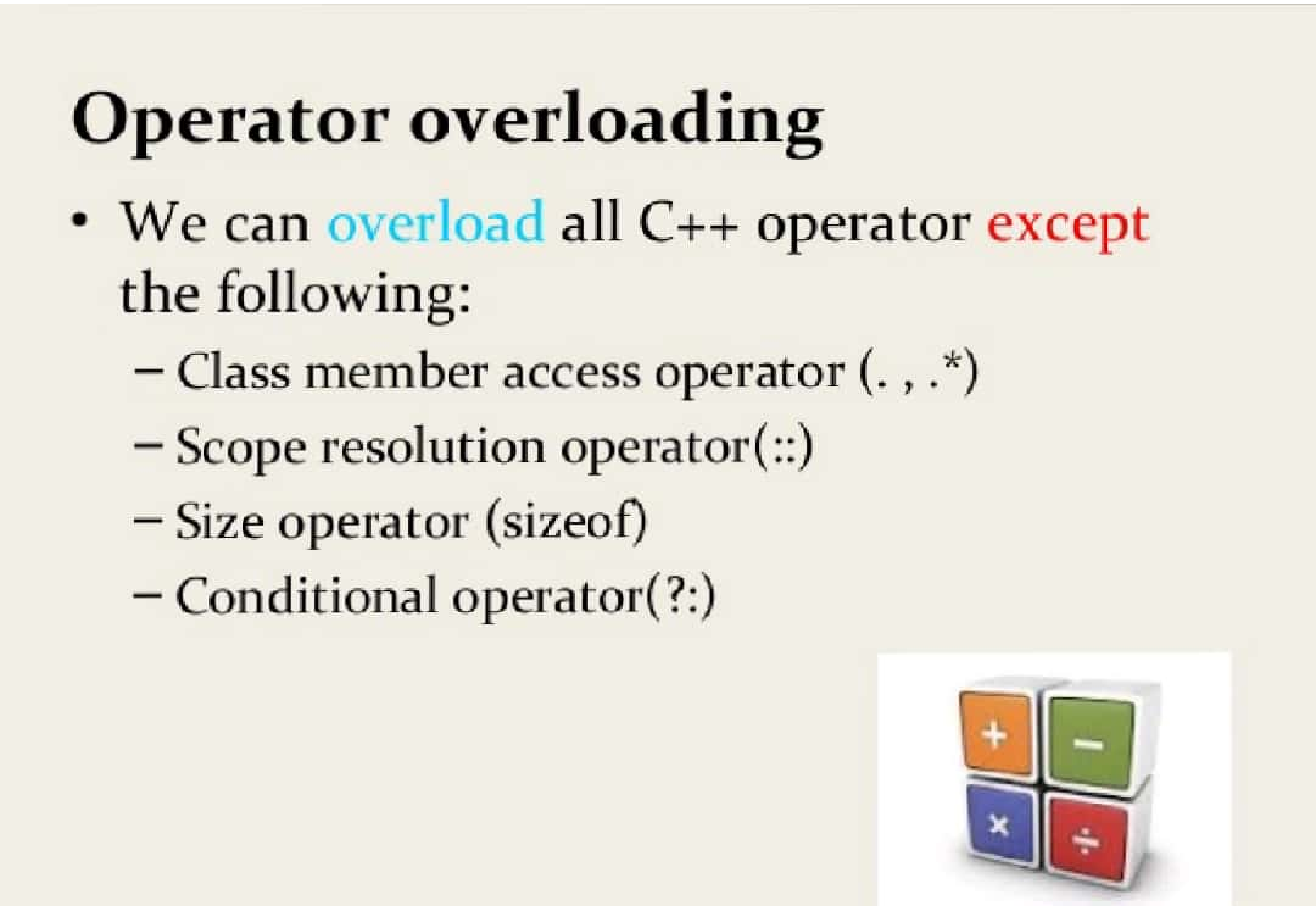
- Addition (+) operator can work on operands of type char, int, float & double.
- However, if s1, s2, s3 are objects of the class string, the we can write the statement,

  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 is known as operator overloading.



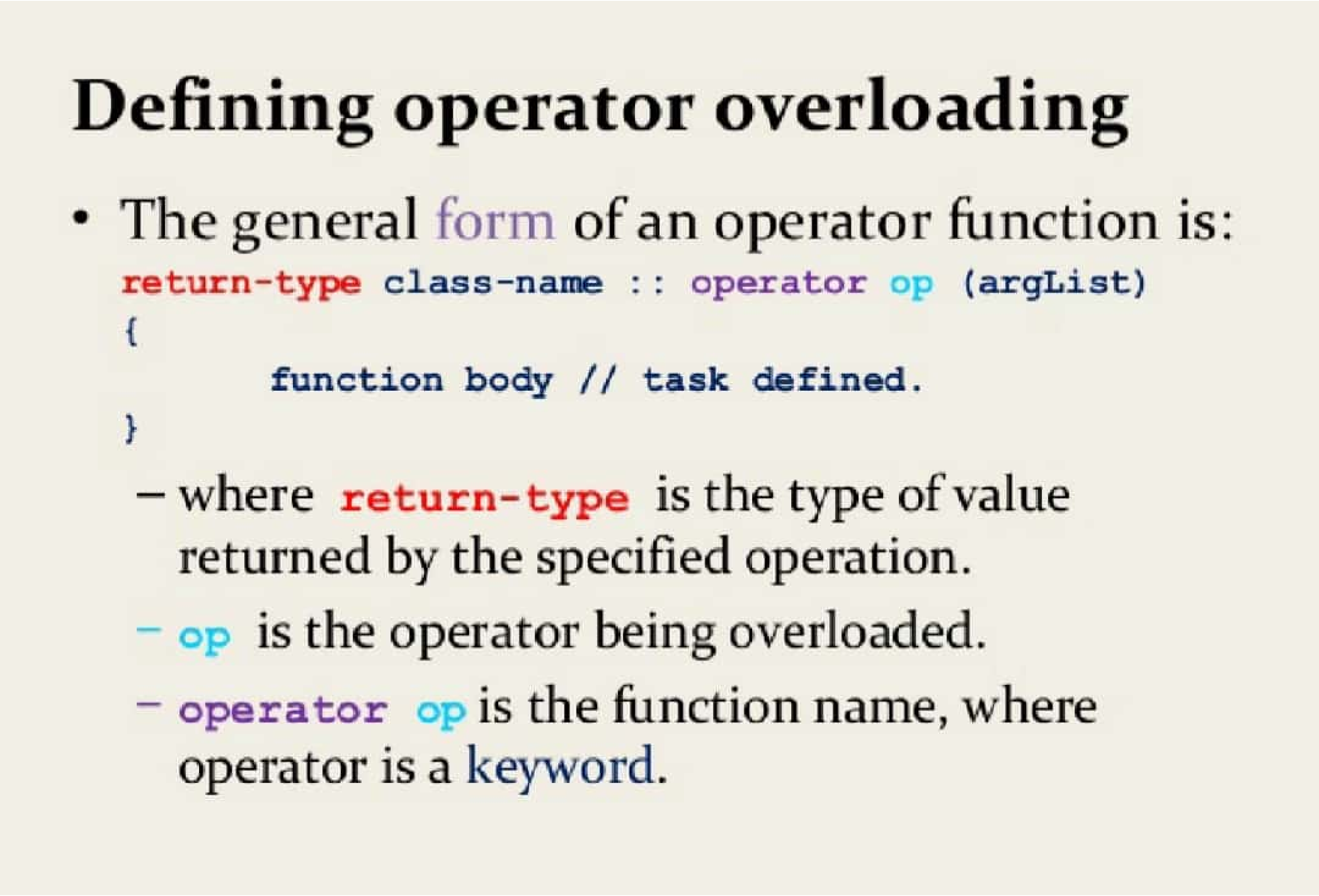
Operator overloading

- 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(?:)

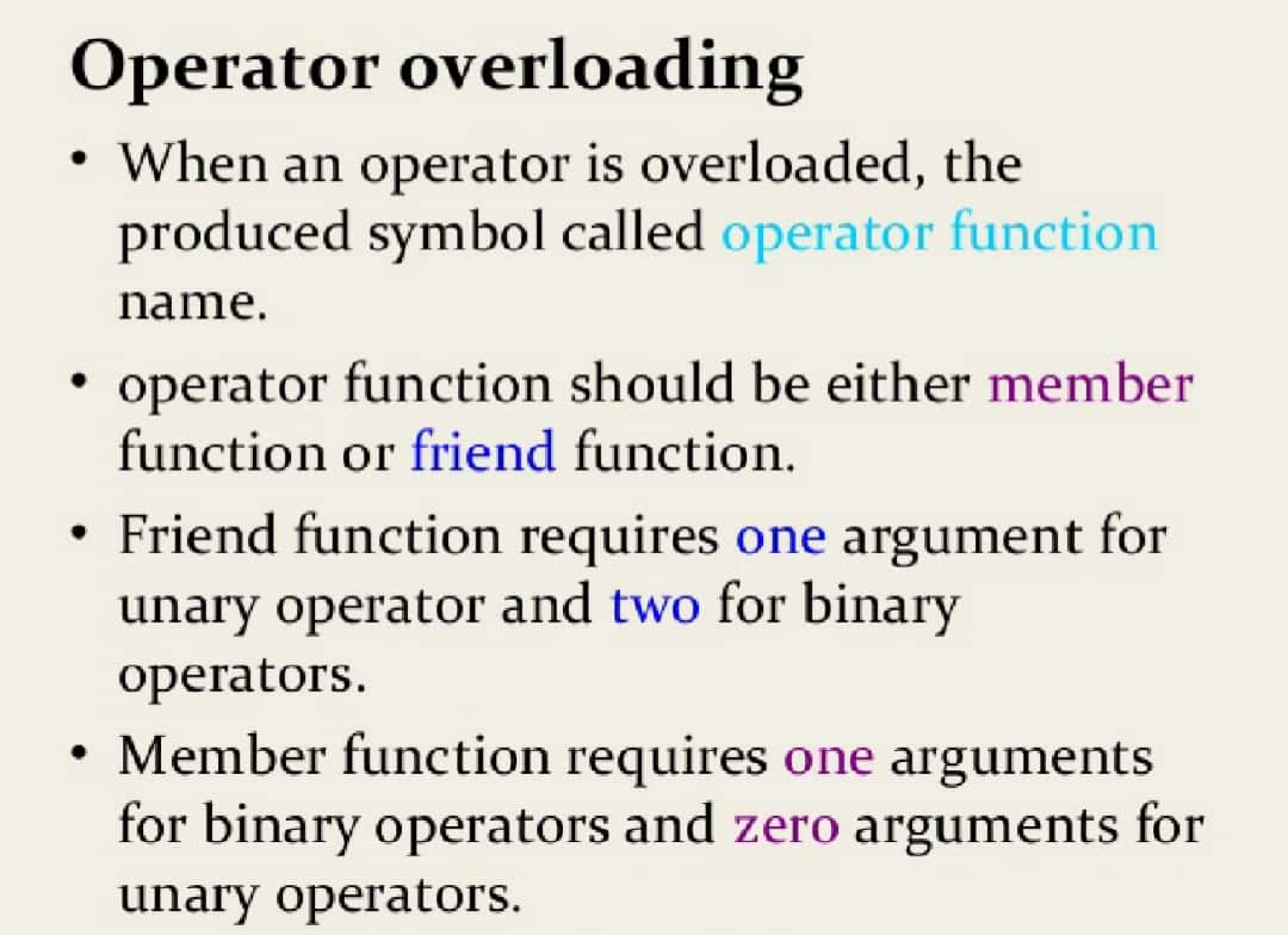


Defining operator overloading

- The general form of an operator function is:

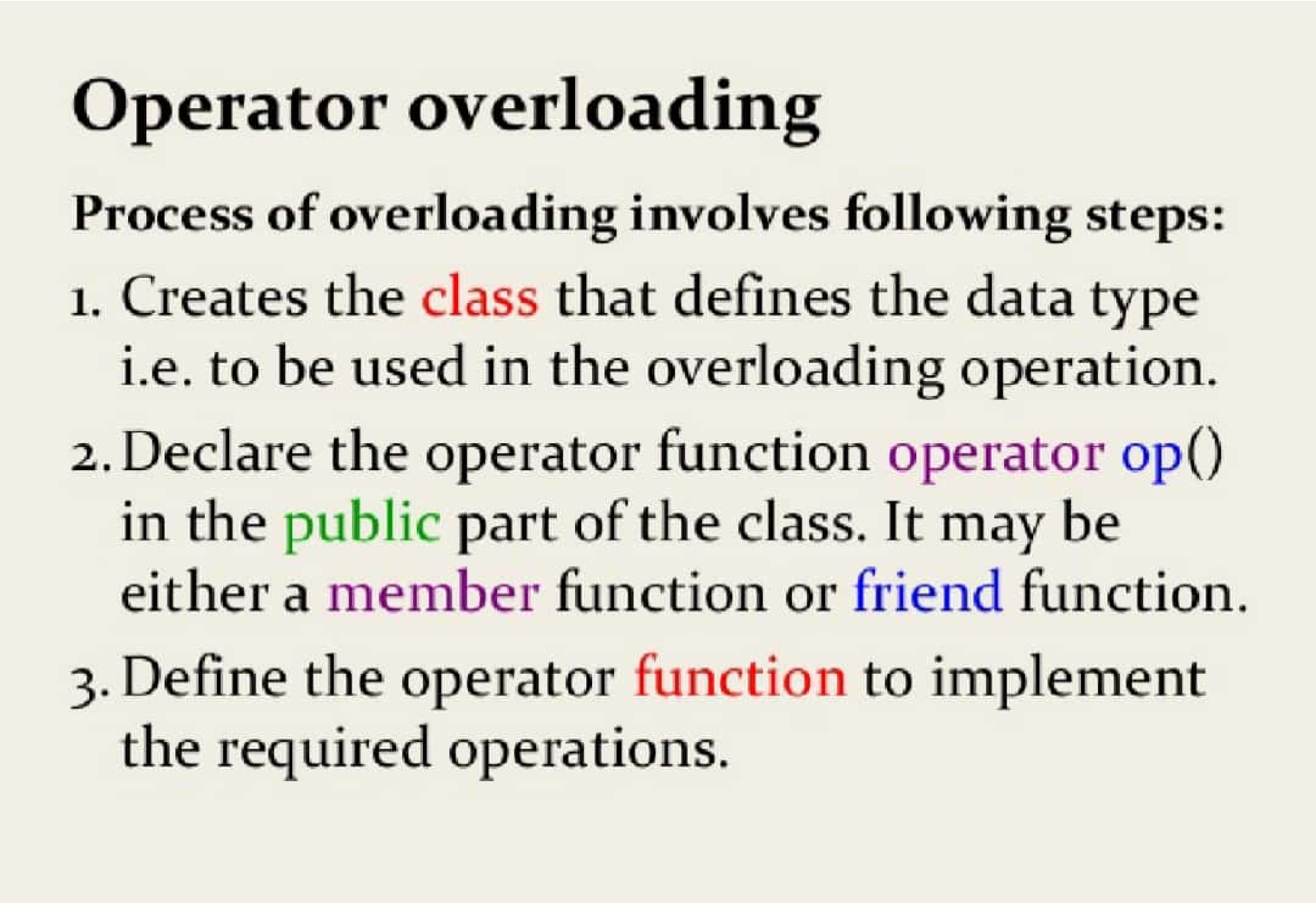
```
return-type class-name :: operator op (argList)
{
        function body // task defined.
}
```

  - 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

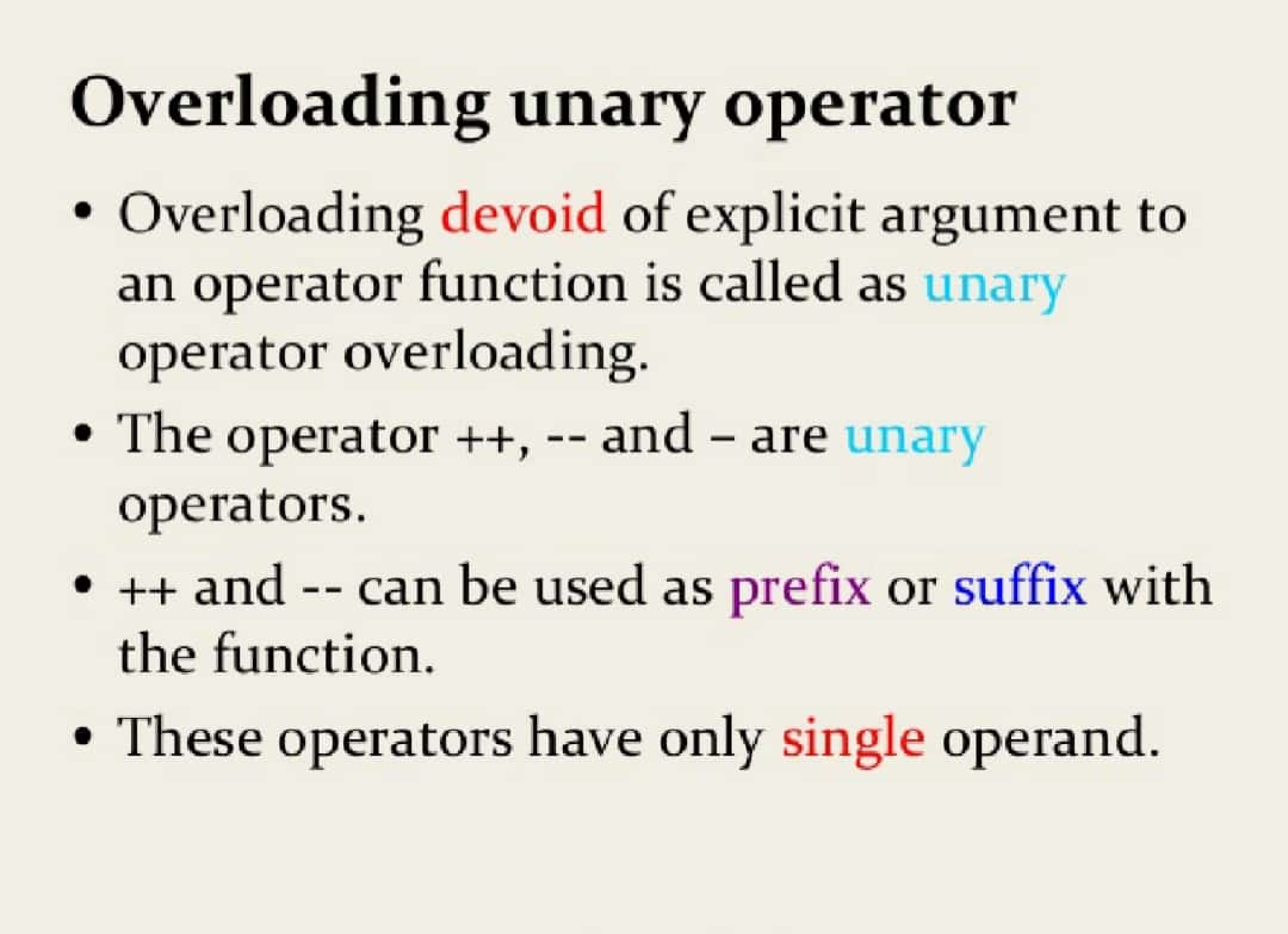
- When an operator is overloaded, the produced symbol called operator function name.
- operator function should be either member function or friend function.
- Friend function requires one argument for unary operator and two for binary operators.
- Member function requires one arguments for binary operators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



Operator overloading

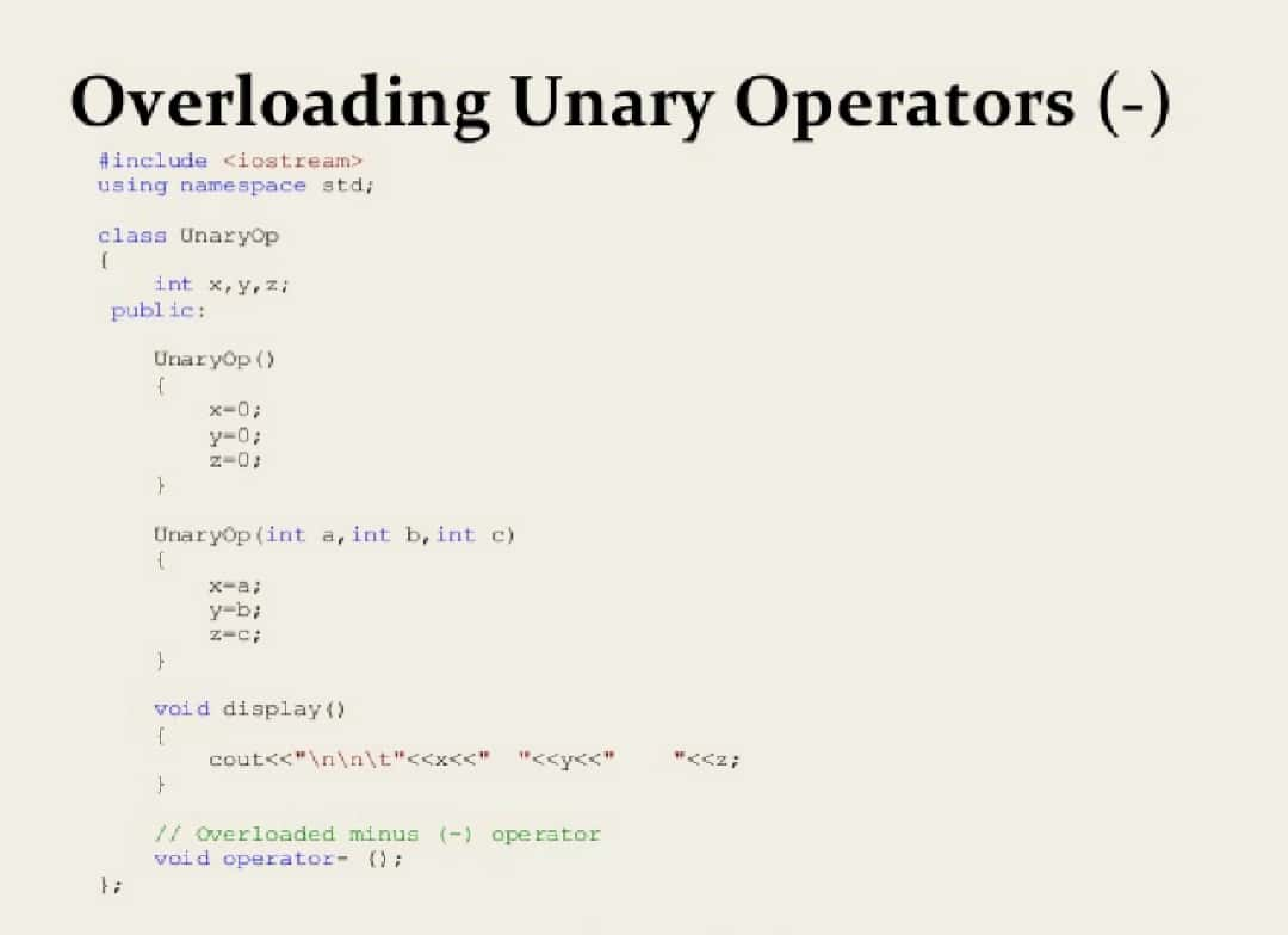
Process of overloading involves following steps:

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

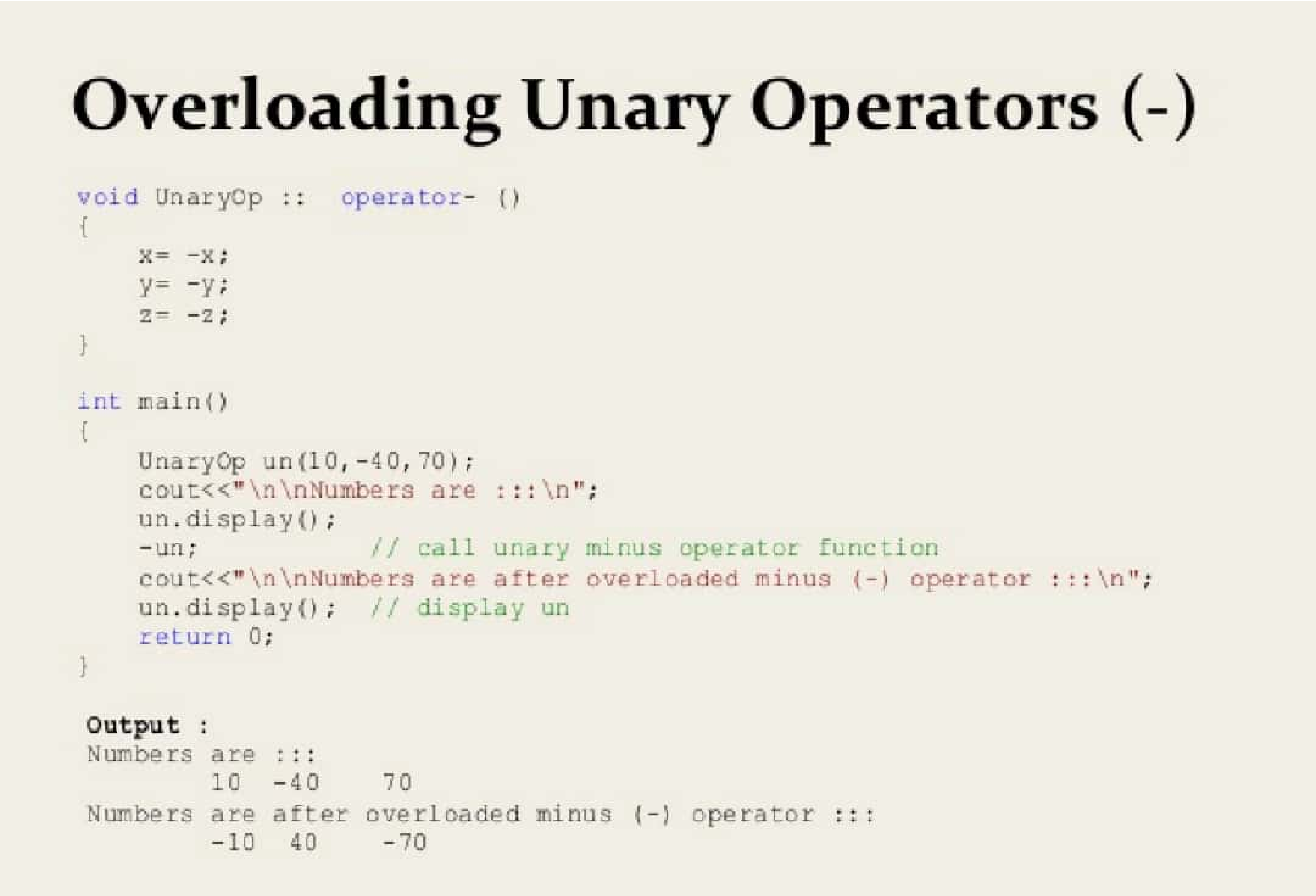
- Overloading devoid of explicit argument to an operator function is called as unary operator overloading.
- The operator ++, -- and – are unary operators.
- ++ and -- can be used as prefix or suffix with the function.
- These operators have only single operand.



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- ();
};
```

Scanned by CamScanner

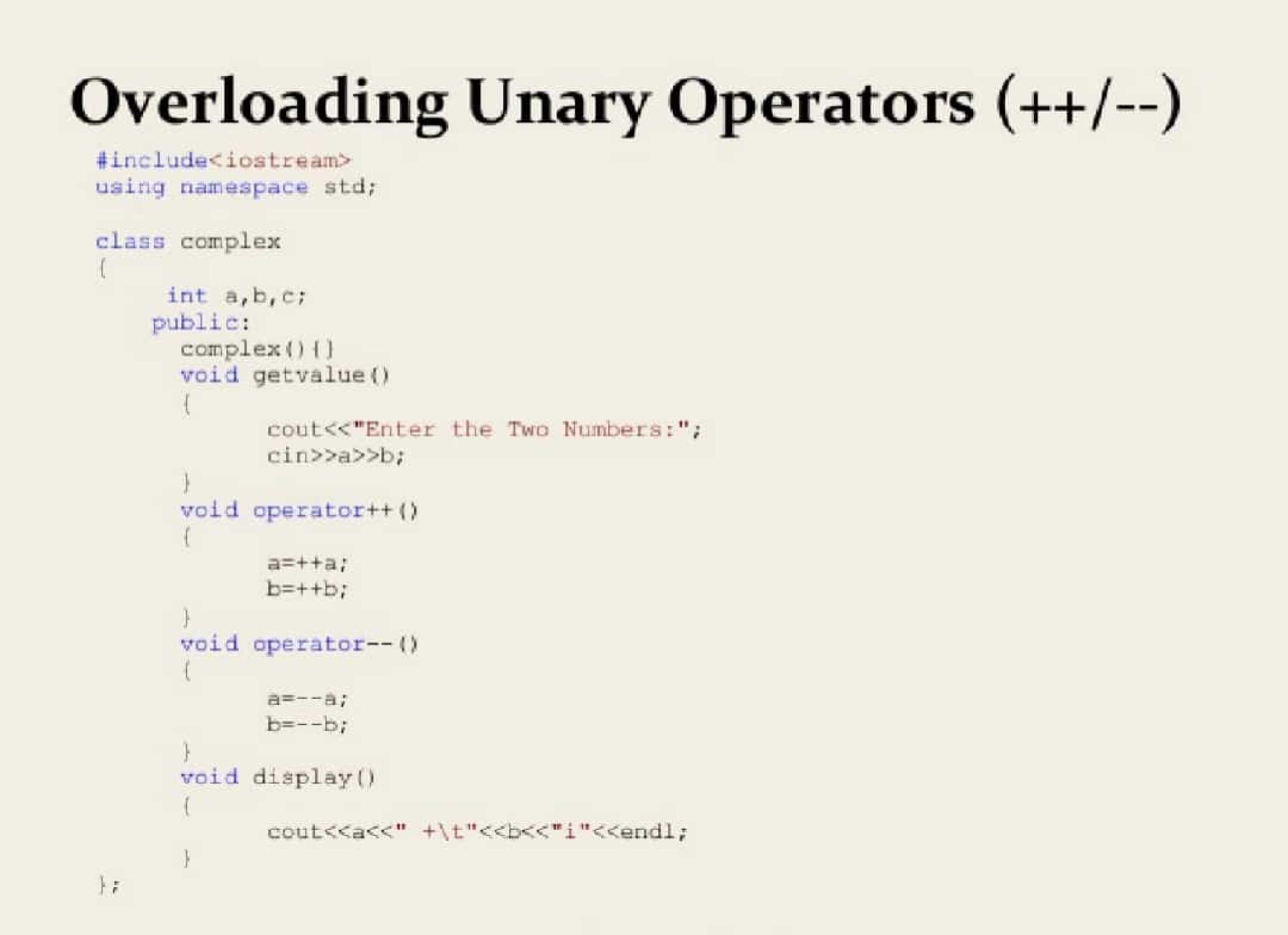


Overloading Unary Operators (-)

```
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;
}
```

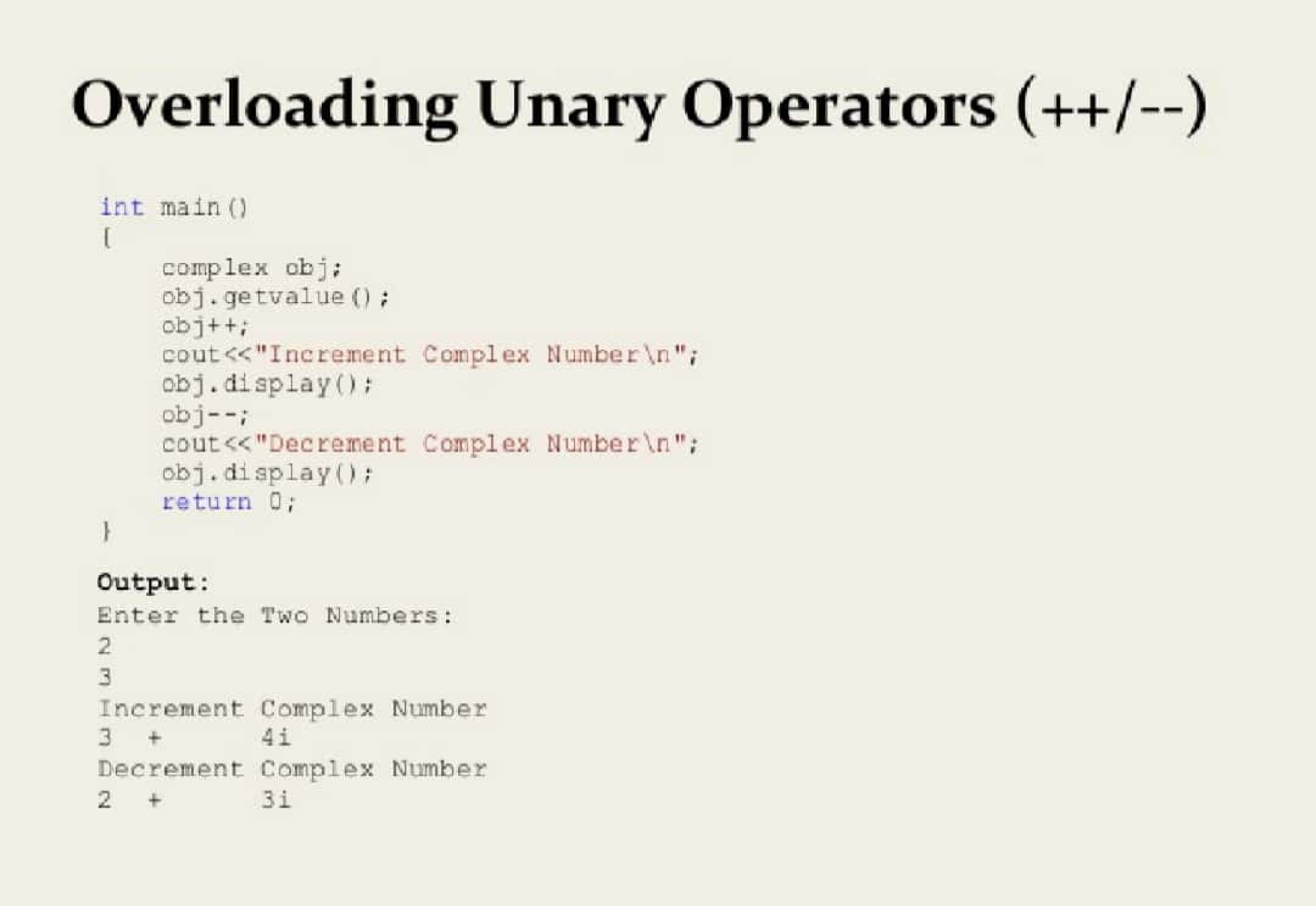
Output :

```
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;
      }
};
```

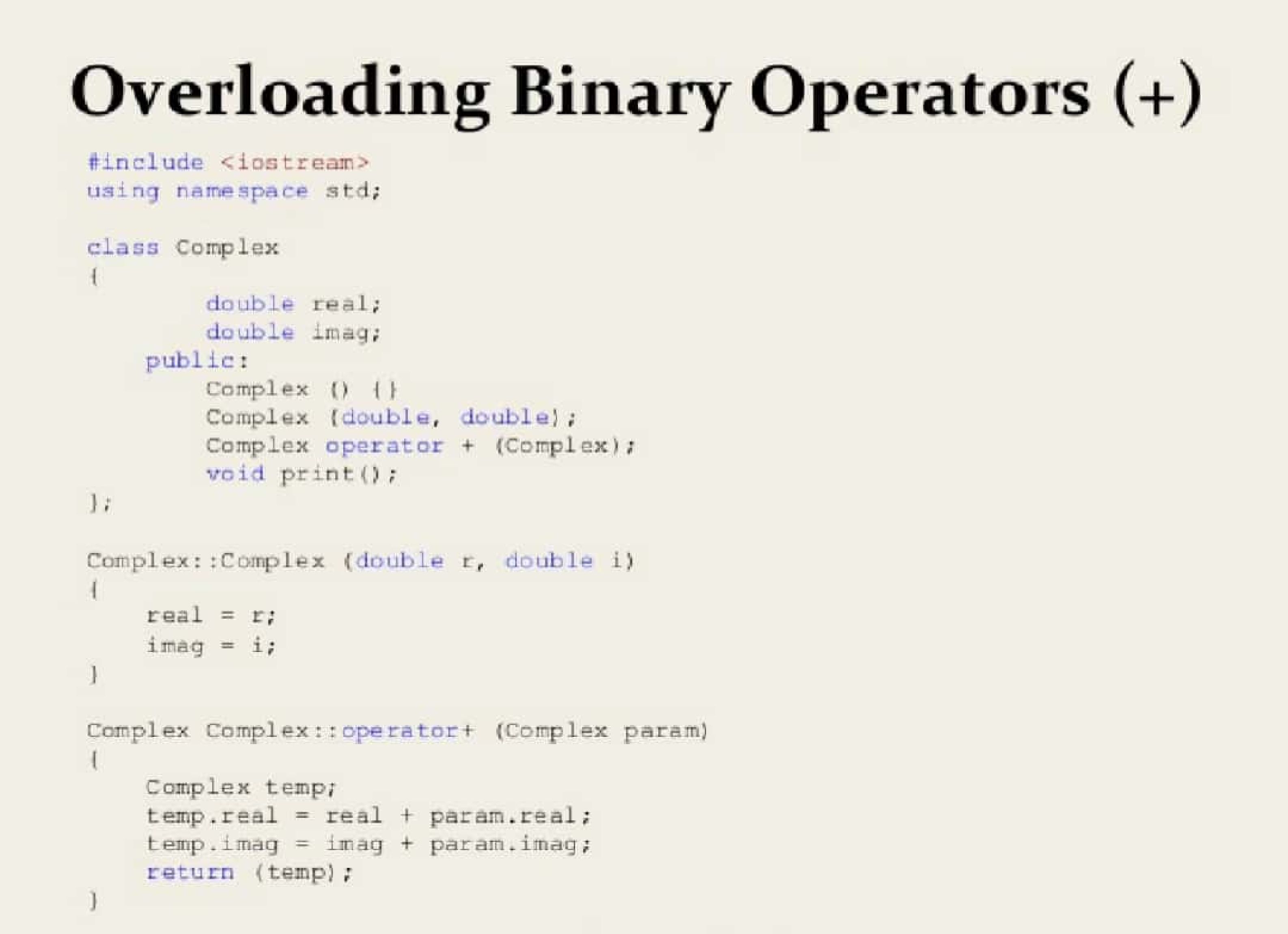


Overloading Unary Operators (++/--)

```
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:

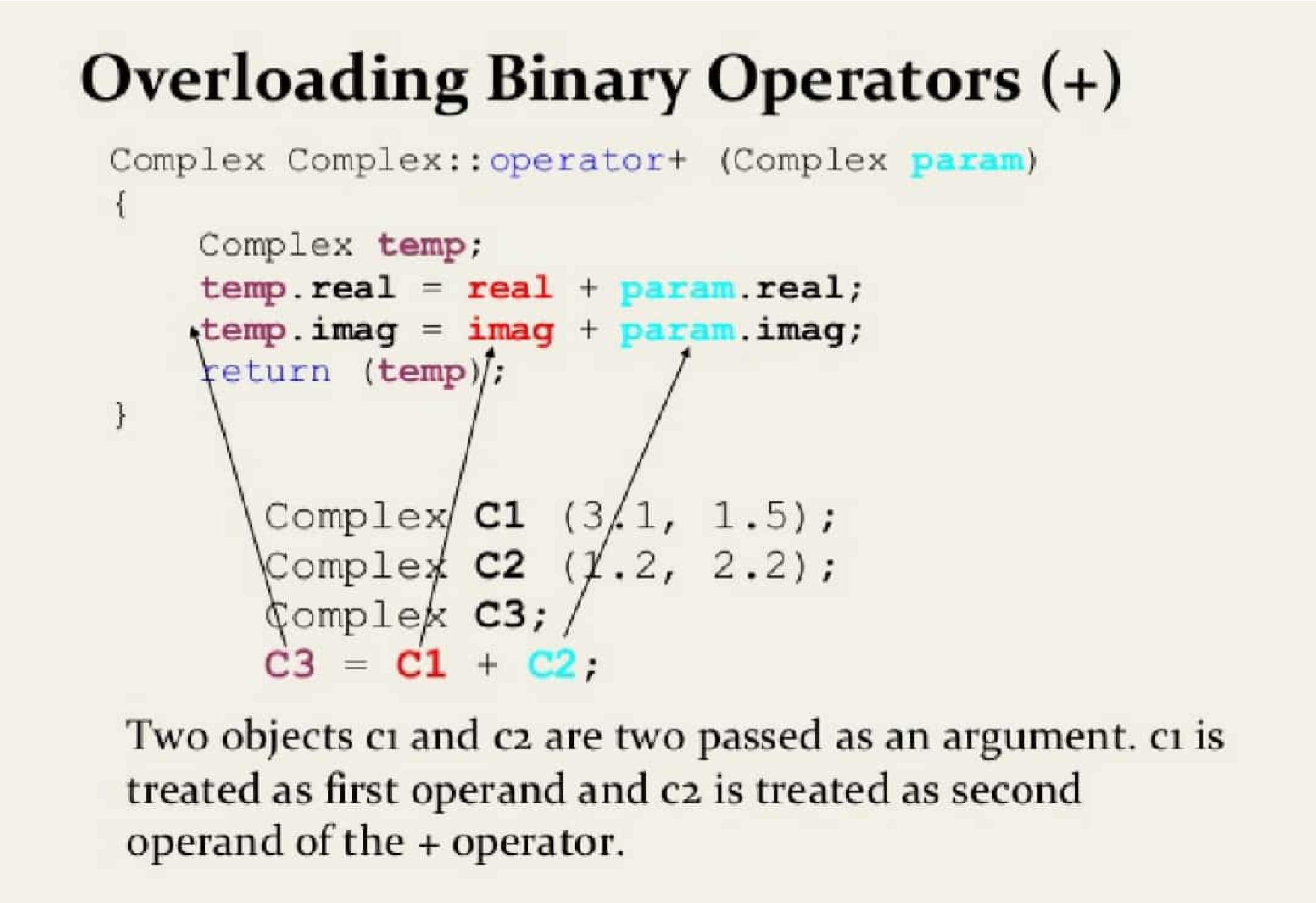
```
Enter the Two Numbers:
2
3
Increment Complex Number
3  +     4i
Decrement Complex Number
2  +     3i
```



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);
}
```

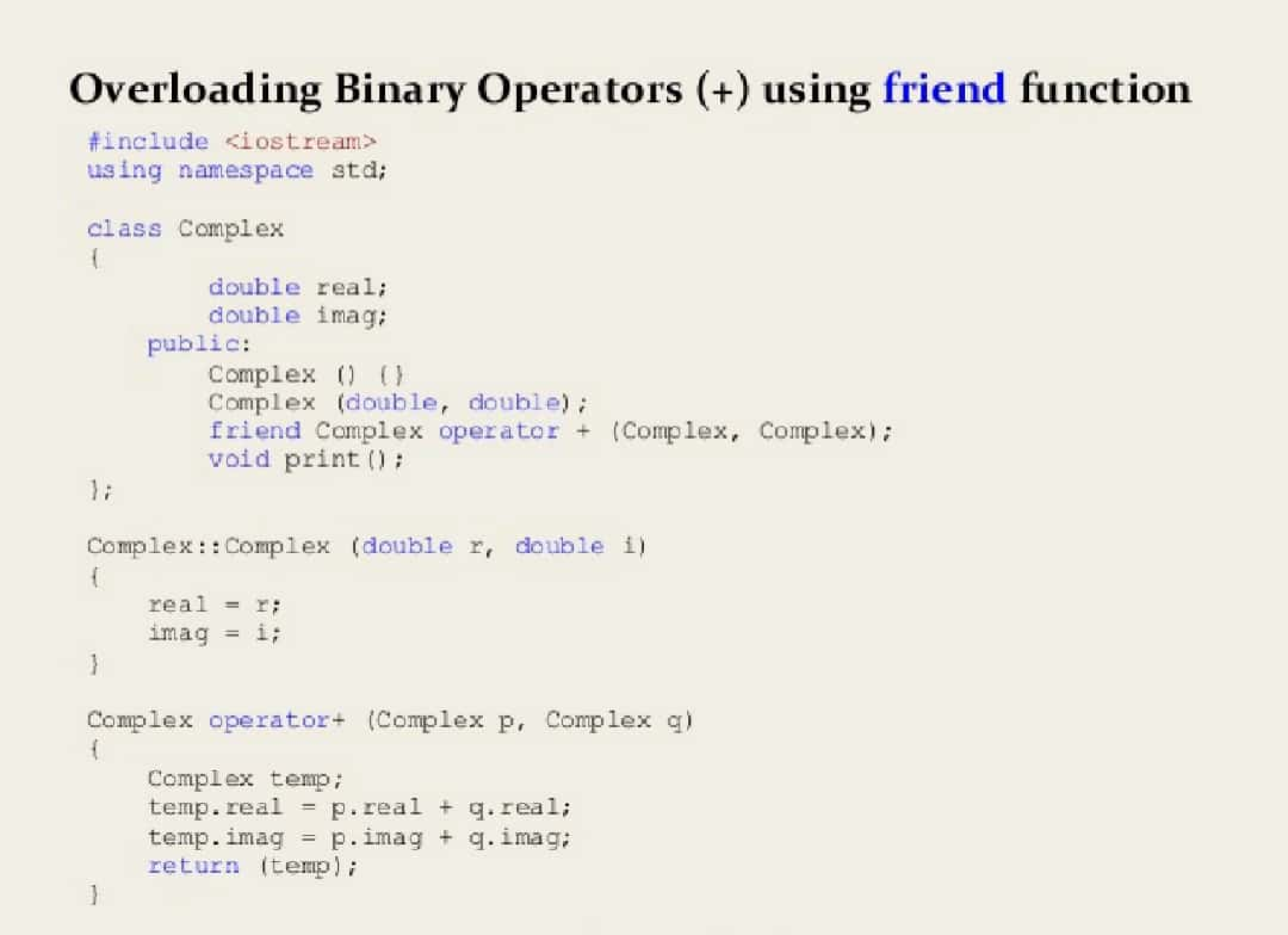
Scanned by CamScanner



Overloading Binary Operators (+)

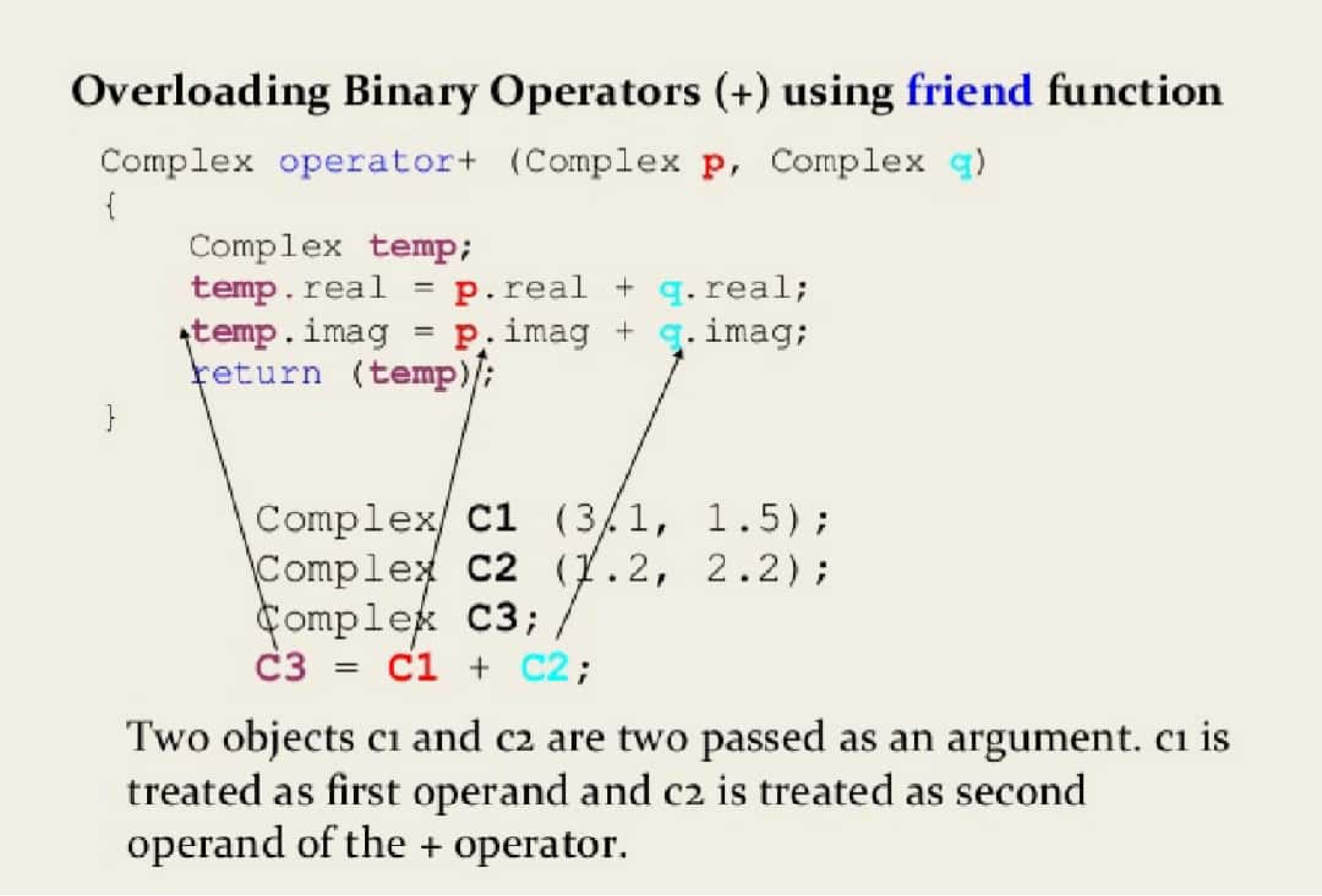
```
Complex Complex::operator+ (Complex param)
{
     Complex temp;
     temp.real = real + param.real;
     temp.imag = imag + param.imag;
     return (temp);
}

         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.



Overloading Binary Operators (+) using friend function

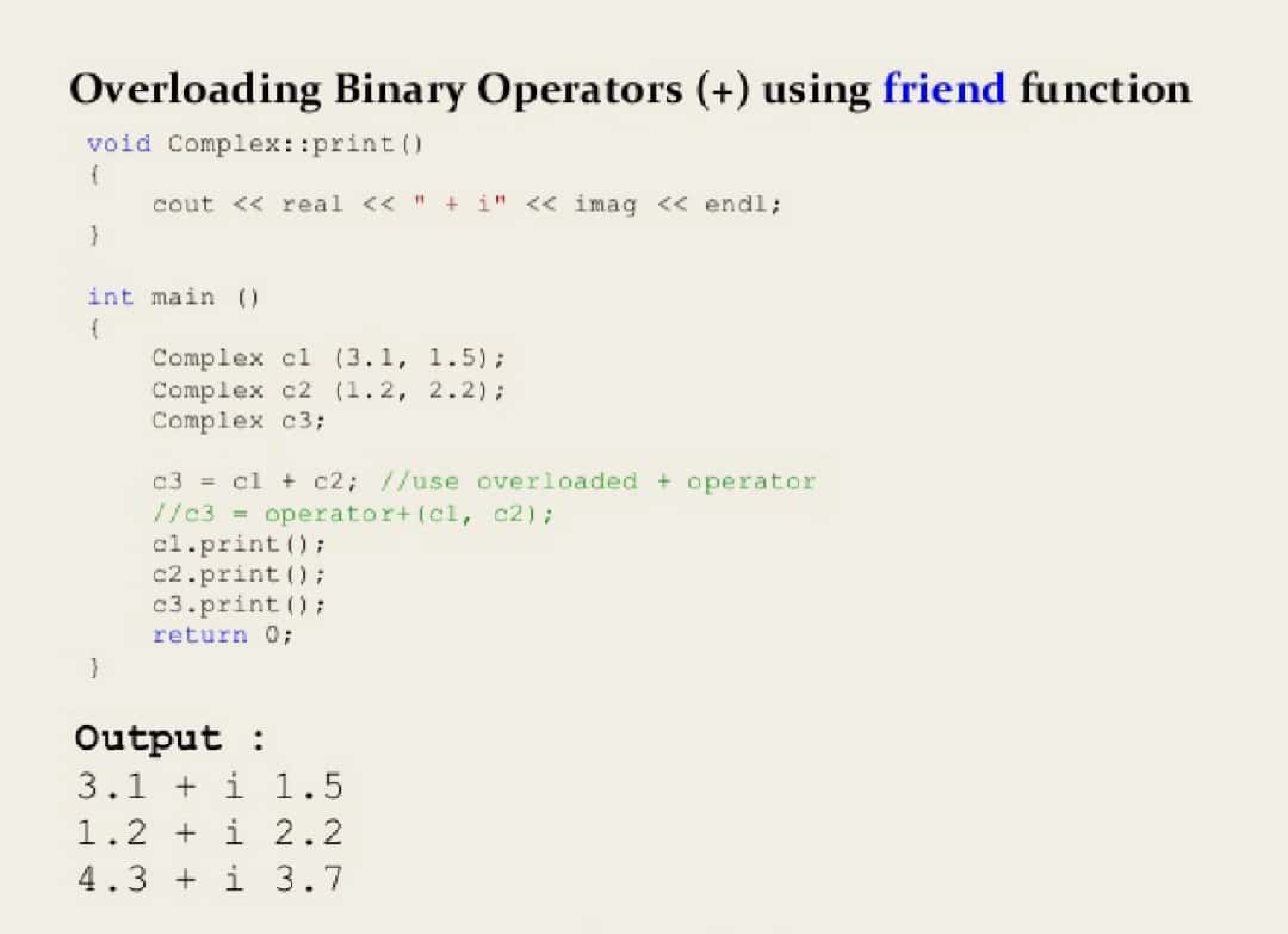
```
#include <iostream>
using namespace std;

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 operand of the + operator.

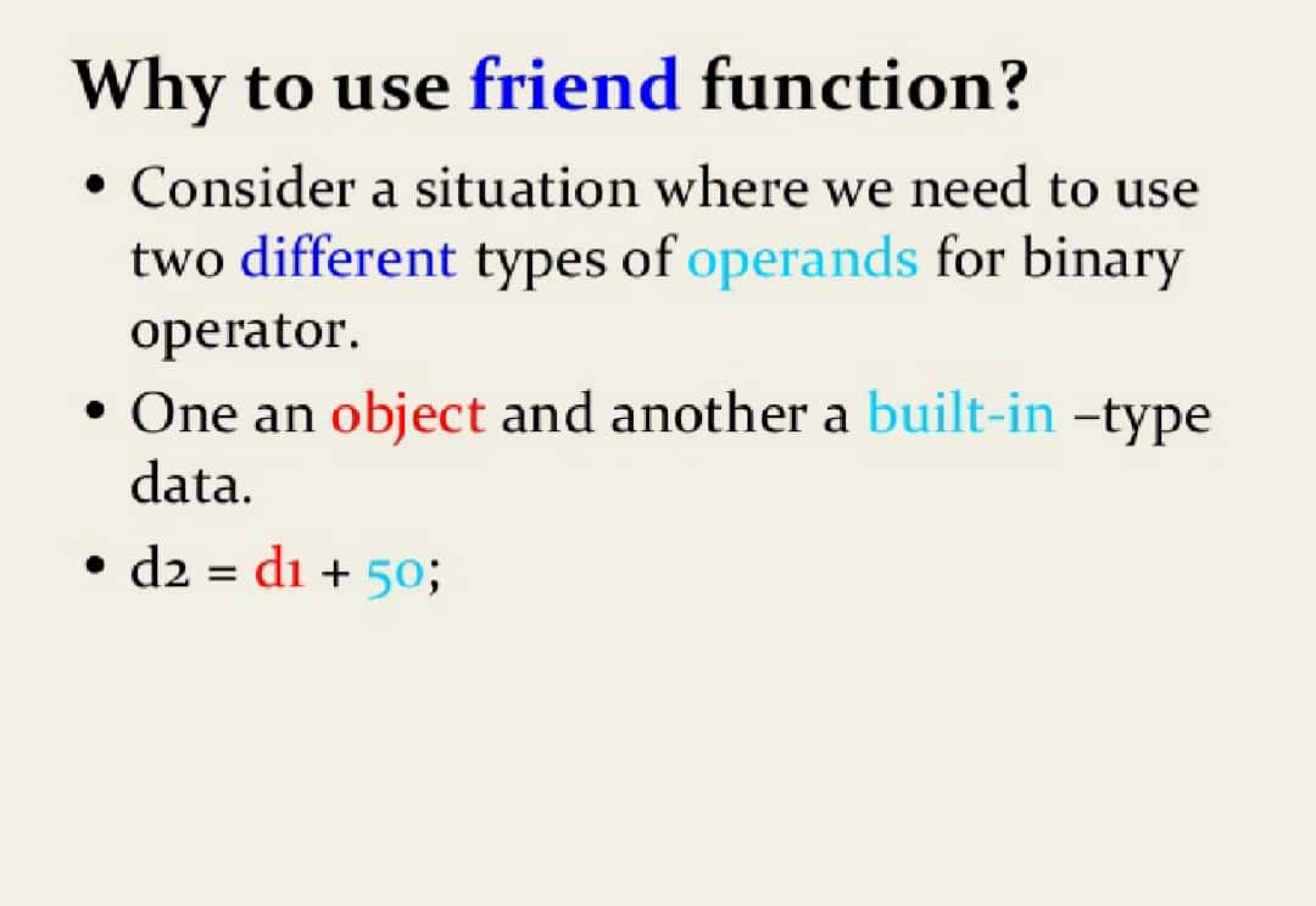


Overloading Binary Operators (+) using friend function

```
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;
}
```

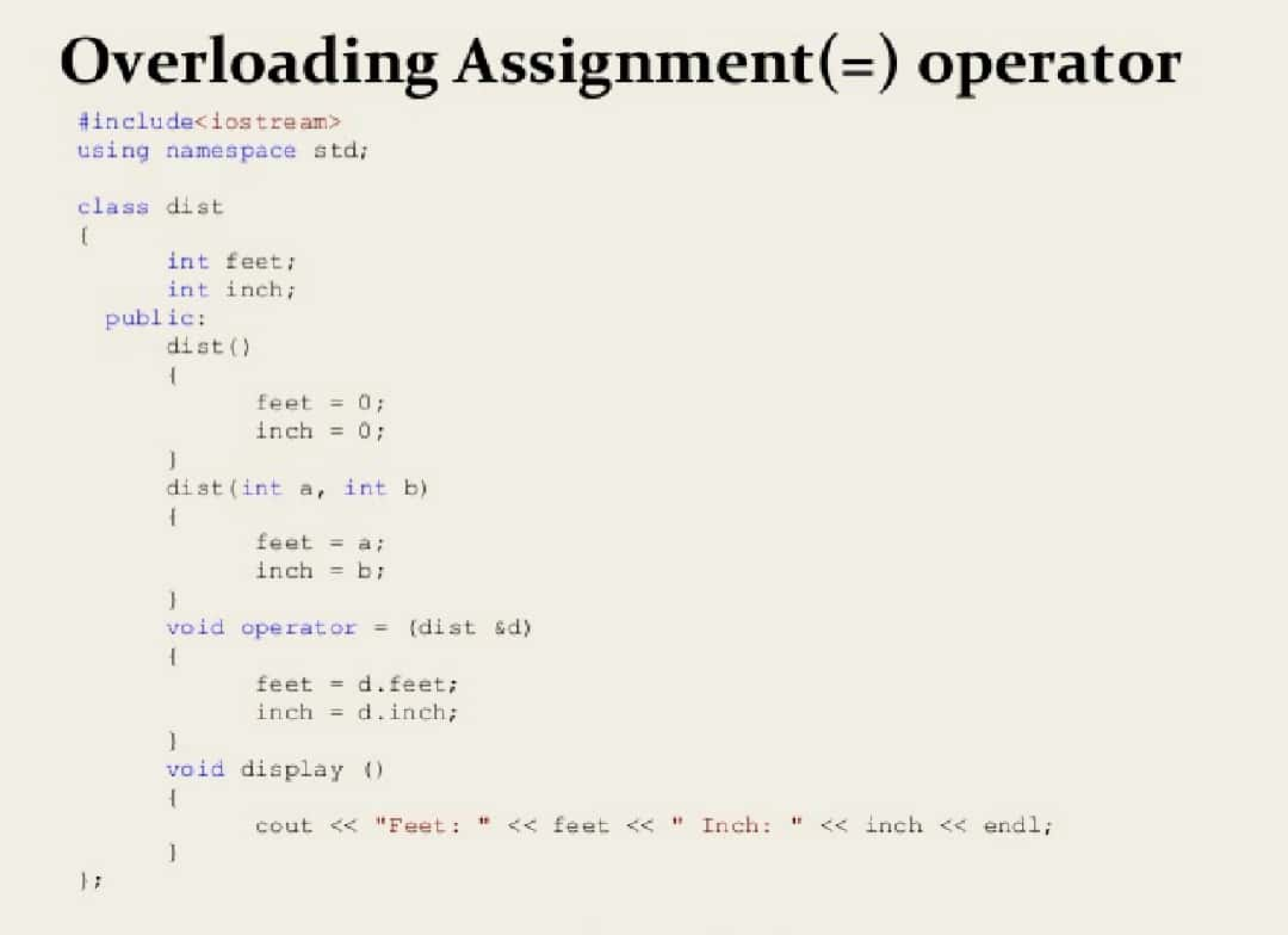
Output :

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



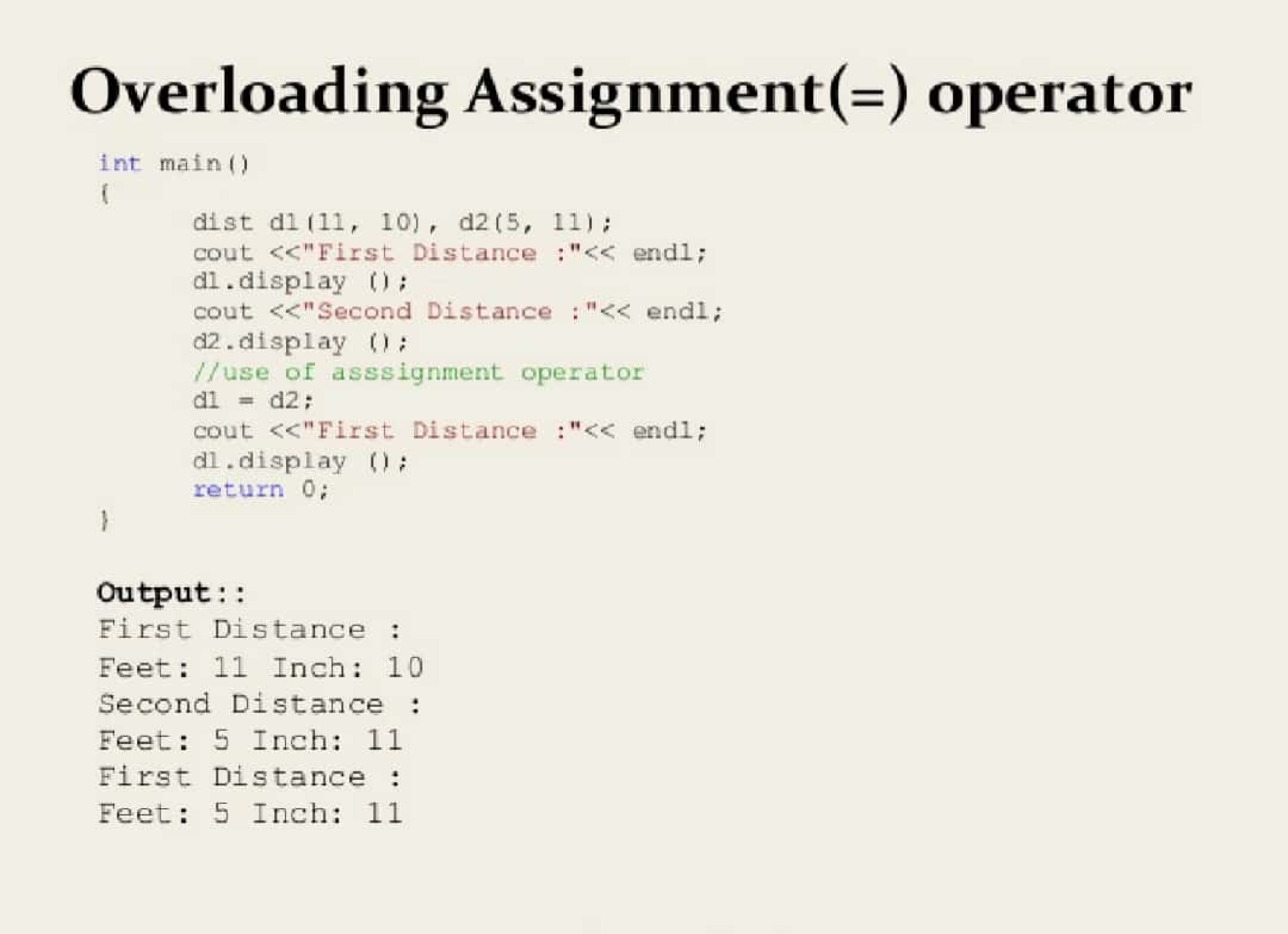
Why to use friend function?

- 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;



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;
        }
};
```

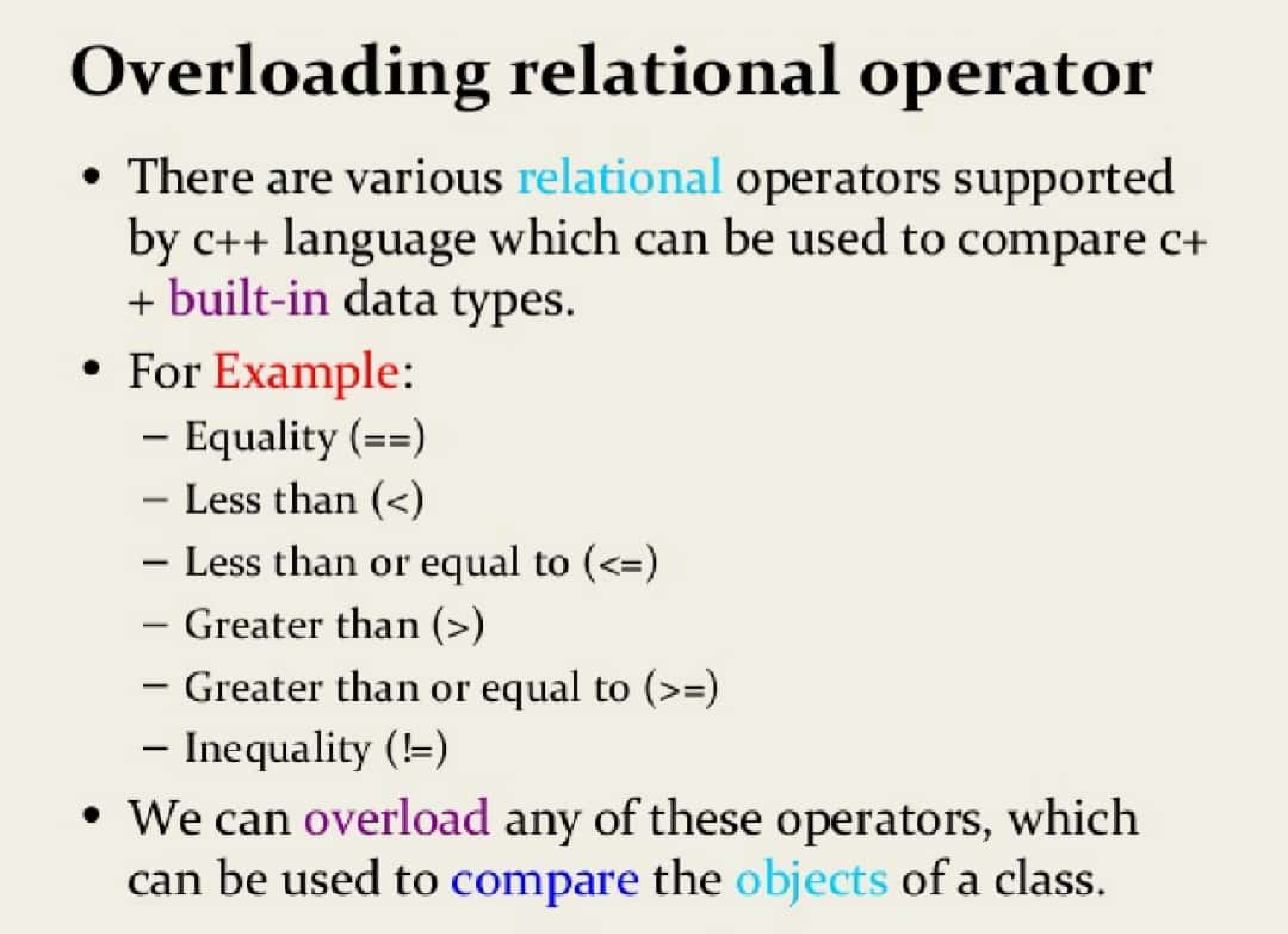


Overloading Assignment(=) operator

```
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;
}
```

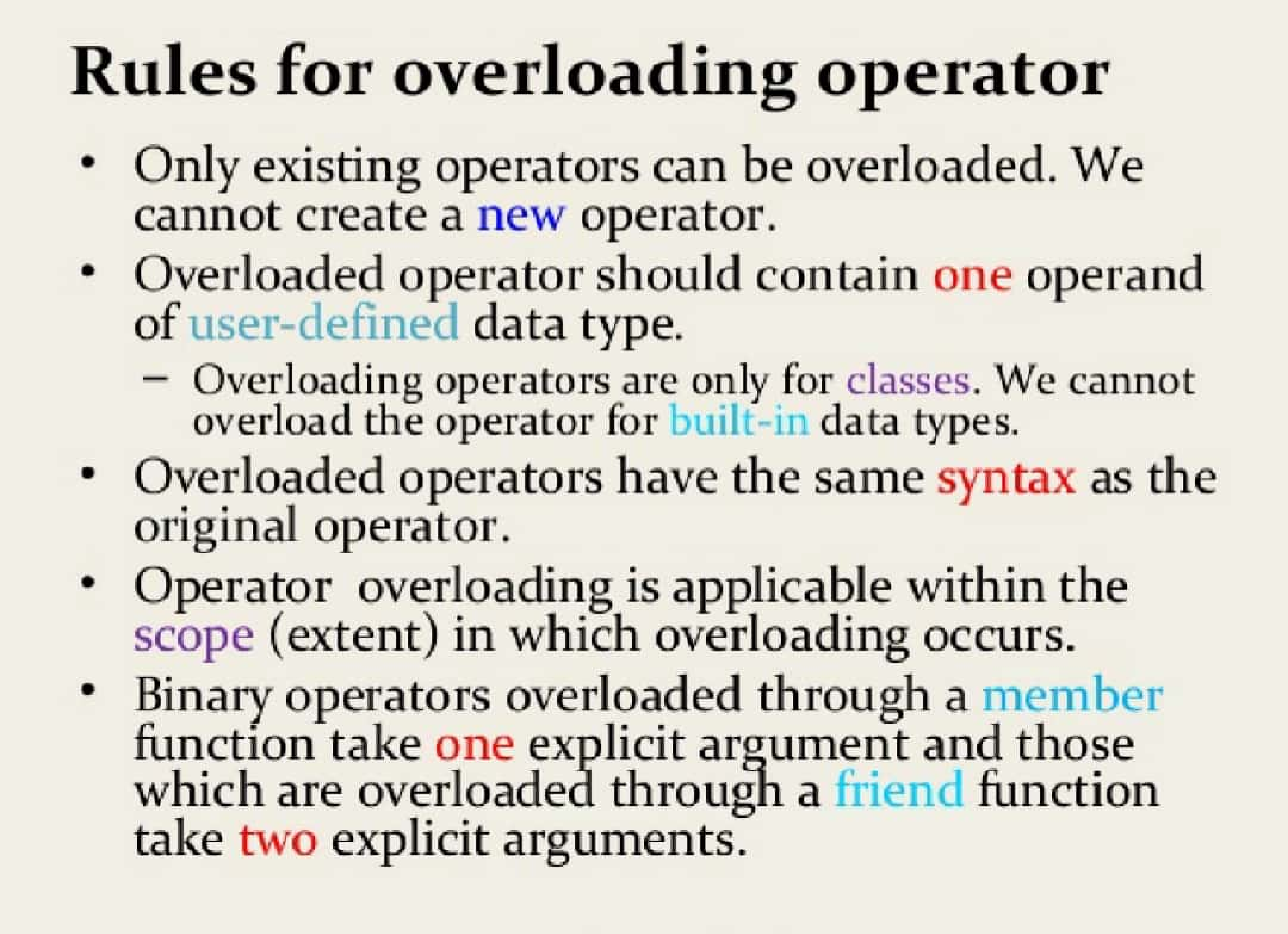
Output::

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



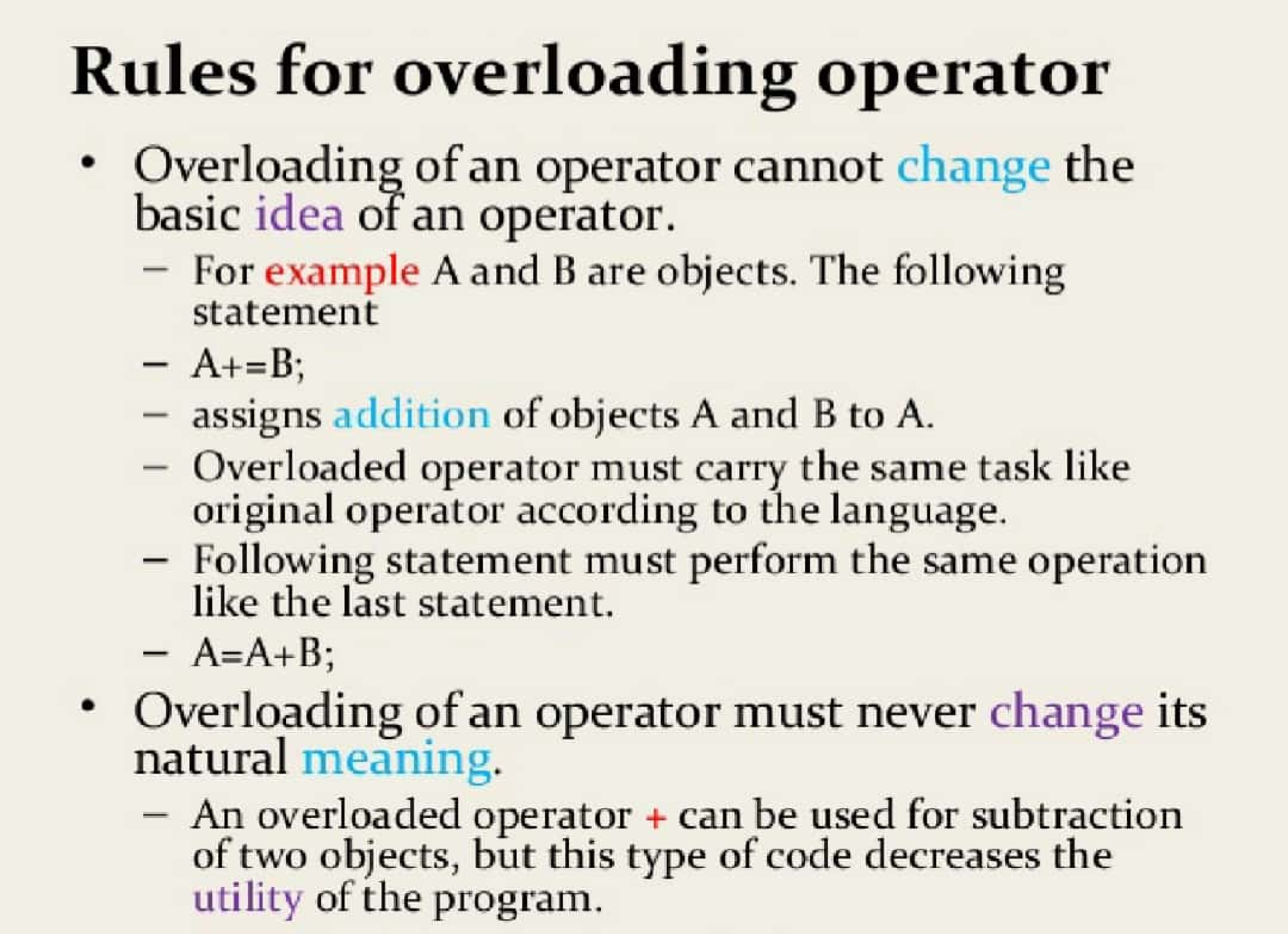
Overloading relational operator

- 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.



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 utility of the program.