Point to ponder in copy constructor

We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like filehandle, a network connection..etc. The default ***constructor does only shallow copy.***

Deep copy is possible only with user defined copy constructor. In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.

```
class String
{
private:
    char *s;
    int size;
public:
void print() { cout << s << endl;
}
     String(const char *str)
{
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}
void change(const char *str)
{
    delete [] s;
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}
~String() { delete [] s;  }
```

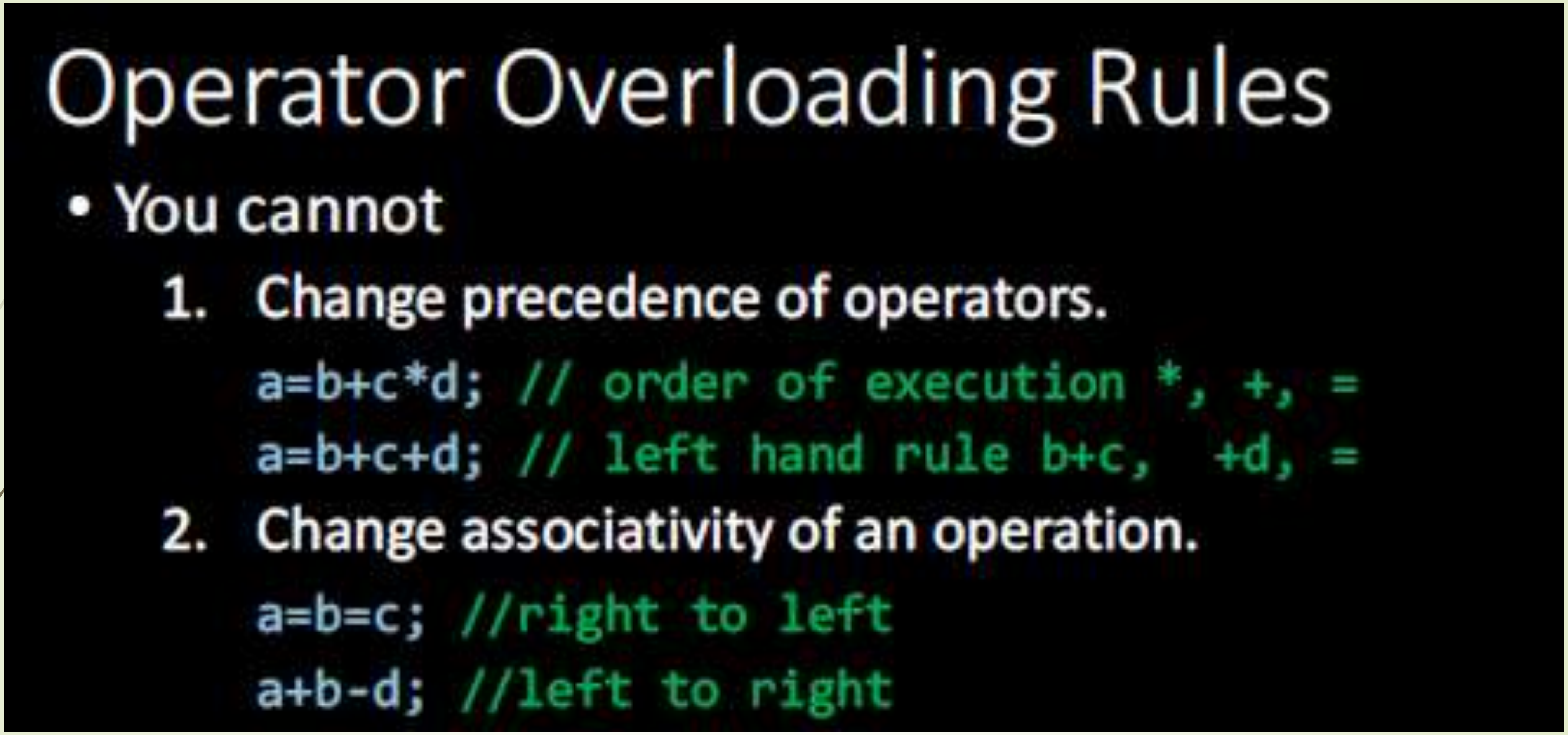
```
String(const String& old_str)
{
    size = old_str.size;
    s = new char[size+1];
    strcpy(s, old_str.s);
}
};
int main()
{
    String str1("Quiz");
    String str2 = str1;

    str1.print(); // what is printed ?
    str2.print();

    str2.change(“QQQuiz");

    str1.print(); // what is printed now ?
    str2.print();
    return 0;
```

C++ Operator Overloading

- In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**.
- Perform operations on class objects as performed on system defined datatypes.

For Example:

```
cout << myobj;
myobj == otherobj;
myobj++;
myobj = otherobj +3;
```



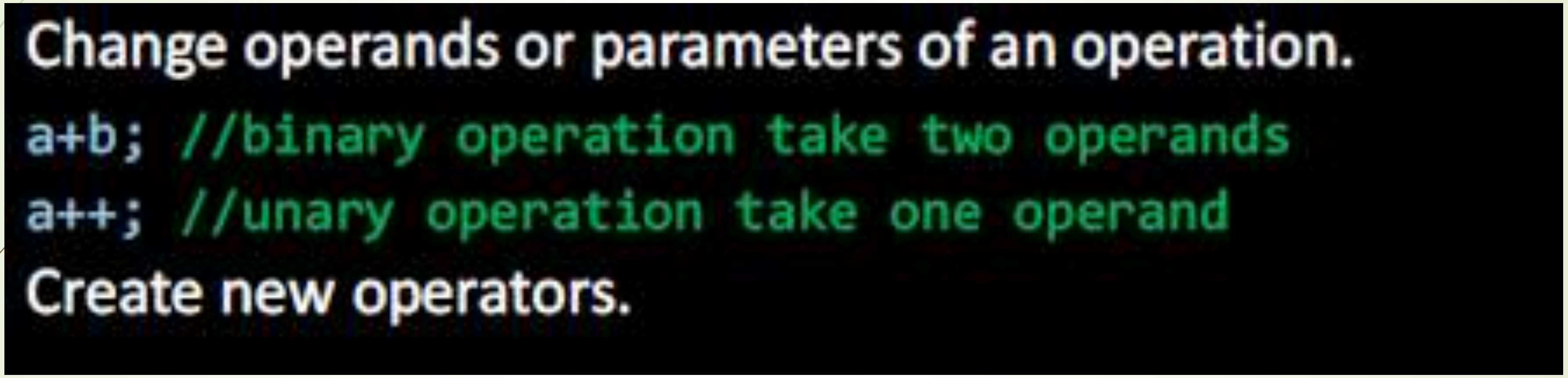
Operator Overloading Rules

- You cannot
  1. Change precedence of operators.

     ```
     a=b+c*d; // order of execution *, +, =
     a=b+c+d; // left hand rule b+c,  +d, =
     ```

  2. Change associativity of an operation.

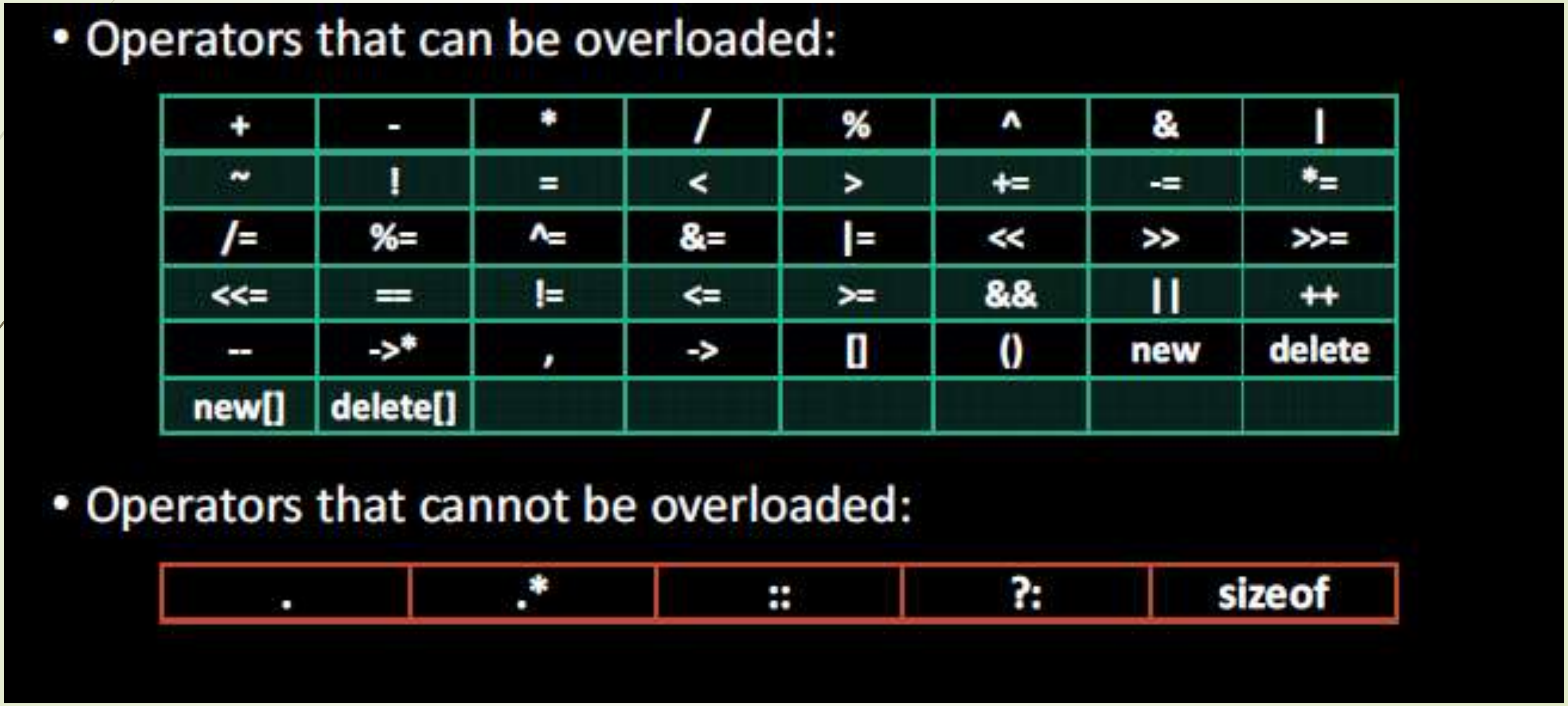
     ```
     a=b=c; //right to left
     a+b-d; //left to right
     ```


Change operands or parameters of an operation.

```
a+b; //binary operation take two operands
a++; //unary operation take one operand
```

Create new operators.

Operators that can be overloaded

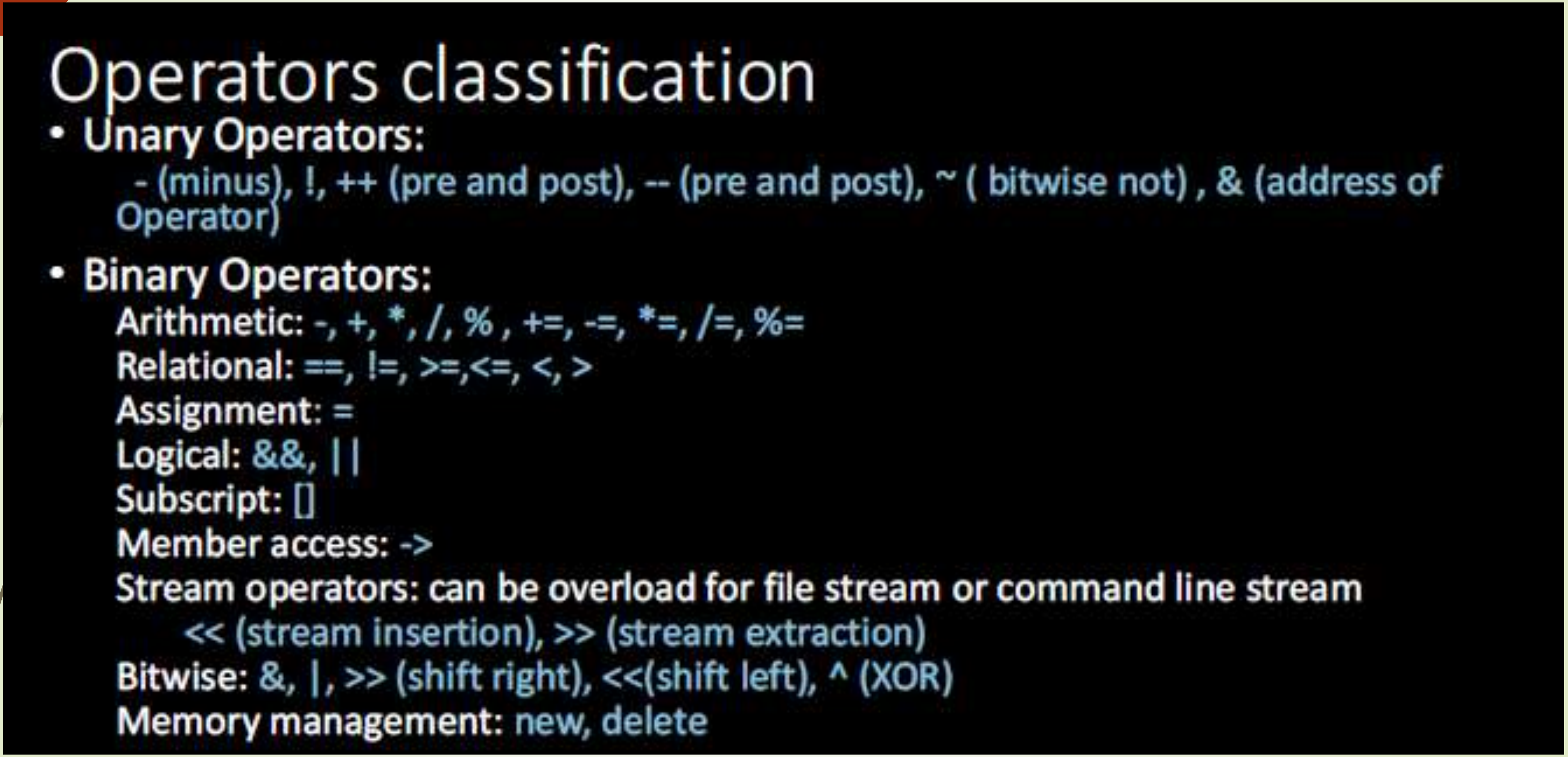


- Operators that can be overloaded:

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

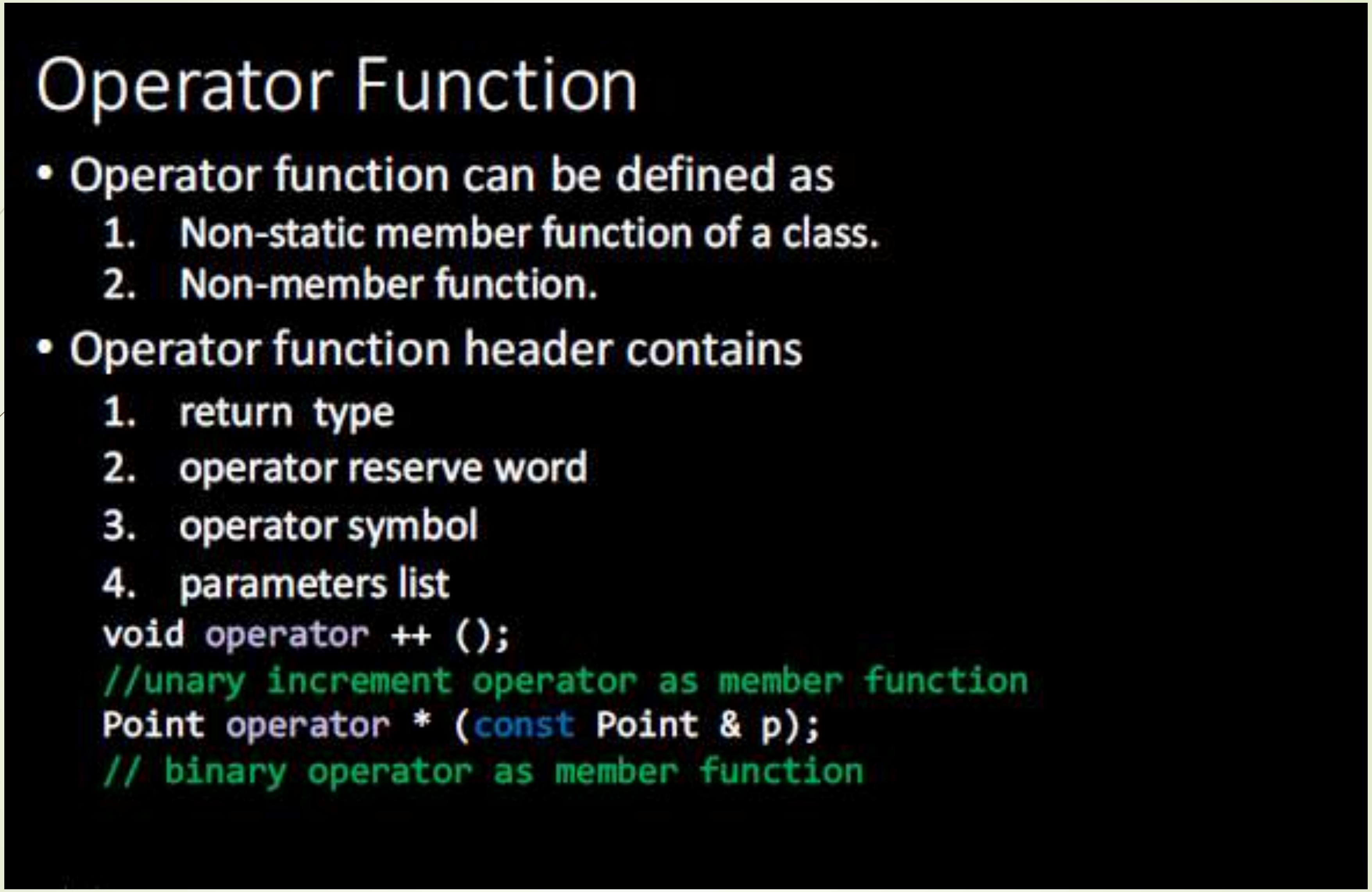
- Operators that cannot be overloaded:

.	.*	::	?:	sizeof



Operators classification

- **Unary Operators:**
  - (minus), !, ++ (pre and post), -- (pre and post), ~ (bitwise not) , & (address of Operator)
- **Binary Operators:**
  Arithmetic: -, +, *, /, % , +=, -=, *=, /=, %=
  Relational: ==, !=, >=,<=, <, >
  Assignment: =
  Logical: &&, ||
  Subscript: []
  Member access: ->
  Stream operators: can be overload for file stream or command line stream
  << (stream insertion), >> (stream extraction)
  Bitwise: &, |, >> (shift right), <<(shift left), ^ (XOR)
  Memory management: new, delete



Operator Function

- Operator function can be defined as
  1. Non-static member function of a class.
  2. Non-member function.
- Operator function header contains
  1. return type
  2. operator reserve word
  3. operator symbol
  4. parameters list

```
void operator ++ ();
//unary increment operator as member function
Point operator * (const Point & p);
// binary operator as member function
```

Overloading Unary Operators

```
class Counter
{
private:
      unsigned int count; //count
public:
Counter() : count(0) //constructor
      { }
unsigned int get_count() //return count
      { return count; }
void operator ++ () //increment (prefix)
      {
      ++count;
      }
};
```

```
int main()
{
      Counter c1, c2; //define and initialize
      cout << c1.get_count();
      cout << c2.get_count();
      ++c1;
      ++c2;
      ++c2;
      cout << c1.get_count(); cout
      <<c2.get_count() << endl;
      }
```

Overloading Unary Operators

The operator Keyword

The keyword operator is used to overload the ++ operator in this declarator:
void operator ++ ().
The return type (void in this case) comes first, followed by the keyword operator, followed by the operator itself (++), and finally the argument list enclosed in parentheses (which are empty here). This declarator syntax tells the compiler to call this member function whenever the ++ operator is encountered, provided the operand

Operator Return Values

```
class Counter
{
private:
	unsigned int count; //count
public:
Counter() : count(0) //constructor
	{ }
unsigned int get_count() //return count
	{ return count; }
Counter operator ++ () //increment count
{
	++count;
	Counter temp;
	temp.count = count;
	return temp;
}};
```

```
int main()
{
	Counter c1, c2; //define and initialize
	cout << c1.get_count();
	cout << c2.get_count();
	++c1; //c1=1
	c2 = ++c1;
	cout << c1.get_count(); cout
	<<c2.get_count() << endl;
	}
```

Nameless Temporary Objects

```
class Counter
{
private:
	unsigned int count; //count
public:
Counter() : count(0) //constructor
	{ }
unsigned int get_count() //return count
	{ return count; }
Counter operator ++ () //increment count
	{
	++count;
	return Counter(count);

	}
};
```

```
int main()
{
	Counter c1, c2; //define and initialize
	cout << c1.get_count();
	cout << c2.get_count();
	++c1; //c1=1
	c2 = ++c1;
	cout << c1.get_count(); cout
	<<c2.get_count() << endl;
	}
```

Postfix Notation

To make both versions of the increment operator work, we define two overloaded ++ operators,

```
class Counter
{
private:
      unsigned int count; //count
public:
Counter() : count(0) //constructor
      { }
unsigned int get_count() //return count
      { return count; }
Counter operator ++ () //increment count
      {
      ++count;
      return Counter(count);

      }
Counter operator ++ (int) {
      return Counter(count++);
      }
};
```

```
int main()
{
      Counter c1, c2; //define and initialize
      cout << c1.get_count();
      cout << c2.get_count();
      ++c1; //c1=1
      c2 = ++c1;
      cout << c1.get_count(); cout
      <<c2.get_count() << endl;

      c2 = c1++;
      cout << c1.get_count(); cout
      <<c2.get_count() << endl;

      }
```

```
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2
```

Postfix Notation

The only difference is the int in the parentheses. This int isn’t really an argument, and it doesn’t mean integer. It’s simply a signal to the compiler to create the postfix version of the operator.

Overloading Binary Operators

dist3.add_dist(dist1, dist2); By overloading the + operator we can reduce this dense-looking expression to

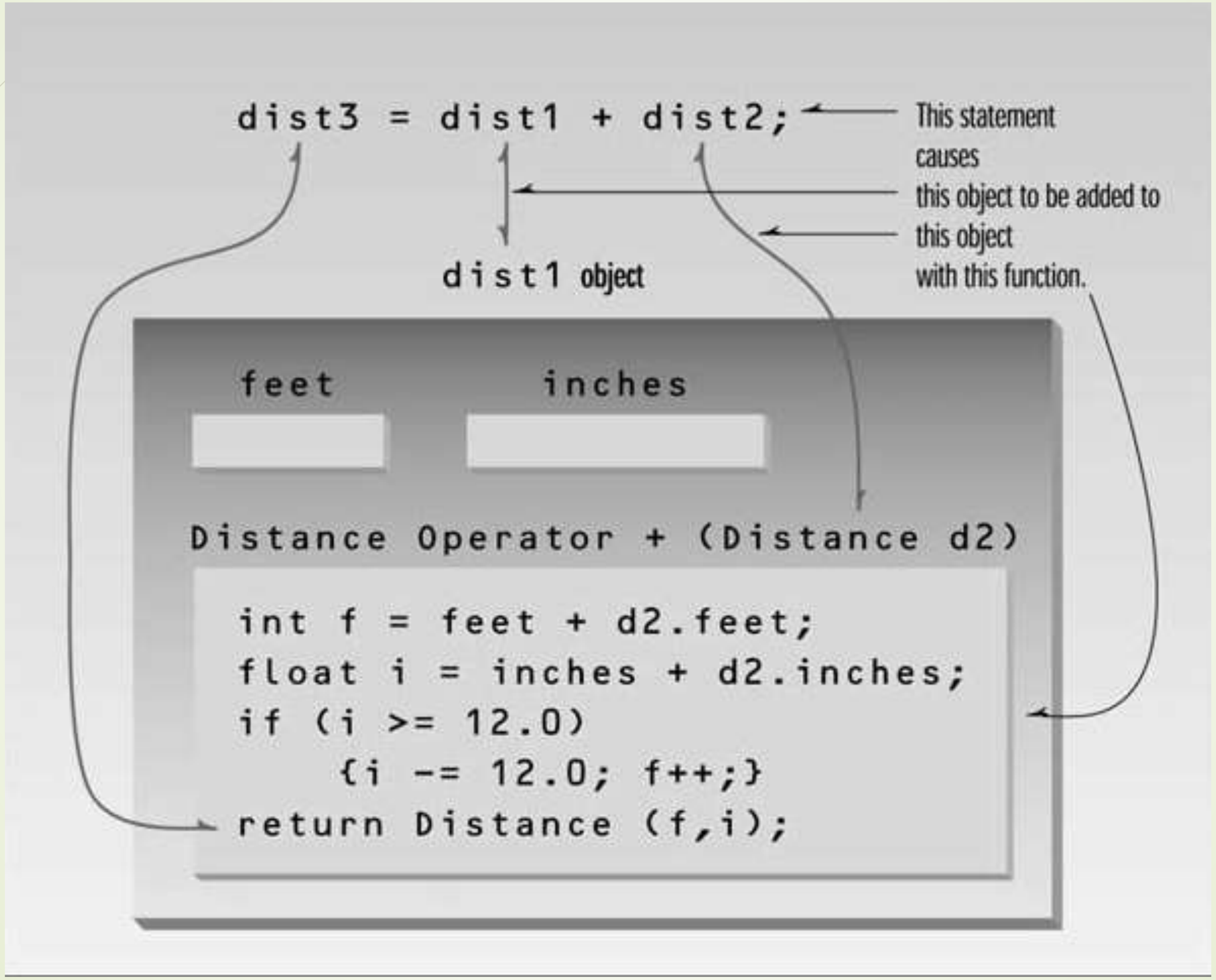
dist3 = dist1 + dist2;

```
Distance operator + ( Distance d2) const{

    int f = feet + d2.feet;
    float i = inches + d2.inches;
    if(i >= 12.0)
        {i -= 12.0; f++;}
    return Distance(f,i);
}
```

Overloading Binary Operators

```
Distance operator + ( Distance d2) {

    int f = feet + d2.feet;
    float i = inches + d2.inches;
    if(i >= 12.0)
        {i -= 12.0; f++;}
    return Distance(f,i);
}   int main()
    {
        Distance dist1, dist3, dist4;
        dist1.getdist();
        Distance dist2(11, 6.25);
        dist3 = dist1 + dist2;
        dist4 = dist1 + dist2 + dist3;
        //display all lengths
        cout << “dist1 = “; dist1.showdist(); cout << endl;
        cout << “dist2 = “; dist2.showdist(); cout << endl;
        cout << “dist3 = “; dist3.showdist(); cout << endl;
        cout << “dist4 = “; dist4.showdist(); cout << endl;
    }
```

Overloading Binary Operators



```
dist3 = dist1 + dist2;
```

```
Distance Operator + (Distance d2)
  int f = feet + d2.feet;
  float i = inches + d2.inches;
  if (i >= 12.0)
      {i -= 12.0; f++;}
  return Distance (f,i);
```

Using this for Returning Values

When you call a member function, it comes into existence with the value of this set to the address of the object for which it was called. The this pointer can be treated like any other pointer to an object, and can thus be used to access the data in the object it points to,

```
class alpha
{
private:
   int data;
public:
   alpha() //no-arg constructor
          { }
   alpha(int d) //one-arg constructor
      { data = d; }
   void display() //display data
   { cout << data; }
   alpha operator = (alpha a) //overloaded = operator
   {
      cout << "\nAssignment operator invoked";
      data = a.data; //not done automatically

      return *this; //return copy of this alpha
   }
};
```

```
int main()
{
alpha a1(37);
alpha a2, a3;
a3 = a2 = a1; //invoke overloaded =, twice
a2.display(); //display a2
a3.display(); //display a3
cout << endl;
return 0;
}
```

- Since this is a pointer to the object of which the function is a member, *this is that object itself, and the statement returns it by reference.

a2=37

a3=37

Each time the equal sign is encountered in

a3 = a2 = a1;

the overloaded operator=() function is called, which prints the messages. The three objects all end up with the same value.

You usually want to return by reference from overloaded assignment operators, using *this, to avoid the creation of extra objects.

Task

The example shown here is a complete program that shows one way to create your own array class:

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Array {
private:
        int* ptr; //pointer to Array contents
        int size; //size of Array
public:
        Array(int s)
        {
                size = s;
                ptr = new int[s];
        }
        ~Array() { delete[] ptr; }
        Int& operator [] (int j)
                { return *(ptr+j); }
};
////////////////////////////////////////////////////////////////
```

```
int main()
{
const int ASIZE = 10; //size of array
Array arr(ASIZE); //make an array
for(int j=0; j<ASIZE; j++) //fill it with squares
        arr[j] = j*j;
for(j=0; j<ASIZE; j++) //display its contents
        cout << arr[j] << ' ';
        cout << endl;
}
```

Binary Operator == is equal to

```
Point p1(3, 4), p2(3, 2);
p1.operator==(p2); // called on p1
Or
cout << (p1==p2);
```

```
class Point {
int x, y;
public:
Point(int a=0, int b=0) { x=a; y=b;}
bool operator==(const Point&);
};
//implementation
bool Point:: operator==(const Point& p){
if (x == p.x && y == p.y)
return true;
else
return false;
```

Binary Operator != not equal to

```
class Point {
int x, y;
public:
Point(int a=0, int b=0) { x=a; y=b;}
bool operator==(const Point&);
bool operator!=(const Point&);
};
//Reuse == operator function
bool Point:: operator!=(const Point& p){
15
return !((*this) == p) ;
}
Point p1(3, 4), p2(3, 2);
p1.operator!=(p2); // called on p1
Or
cout << (p1!=p2);
```

Binary Operator != not equal to

```
p1=p2; // called on p1
p1=p2=p3; // cascaded call
```

Binary Operator = Assignment

```
class Point {
int x, *y;
public:
Point() { x=0; y=nullptr; }
Point(int a, int b) {
   x=a;
   y=new int(b);
}
Point operator=(const Point& p);
};
```

```
Point Point::operator=(const Point&
p){
if (this != &p) {
   x= p.x;
   if(y==nullptr && p.y!=nullptr)
         y = new int(*(p.y));
   else if(y!=nullptr &&
p.y==nullptr){
         delete y;
         y = nullptr;
      }
else if(y!=nullptr && p.y!=nullptr)
      *y = *(p.y);
   }
return *this;
```

Friend function

- A friend function in C++ is defined as a function that can access private, protected and public members of a class.

The friend function is declared using the friend keyword inside the body of the class.

Friend Function Syntax:

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

By using the keyword, the **‘friend’** compiler understands that the given function is a friend function.

Breaching the Walls

We should note that friend functions are controversial. During the development of C++, arguments raged over the desirability of including this feature. On the one hand, it adds flexibility to the language; on the other, it is not in keeping with *data hiding*, the philosophy that only member functions can access a class’s private data.

How serious is the breach of data integrity when friend functions are used? A friend function must be declared as such within the class whose data it will access. Thus a programmer who does not have access to the source code for the class cannot make a function into a friend. In this respect, the integrity of the class is still protected

Use of Friend function in C++

As discussed, we require friend functions whenever we have to access the private or protected members of a class. This is only the case when we do not want to use the objects of that class to access these private or protected members.

To understand this better, let us consider two classes: Tokyo and Rio. We might require a function, metro(), to access both these classes without any restrictions. Without the friend function, we will require the object of these classes to access all the members.

```
class Distance //English Distance class
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0) { }
Distance(float fltfeet) {
    feet = fltfeet;
    inches = 12*(fltfeet-feet);}
Distance(int ft, float in) { feet = ft; inches = in; }
void showdist()
{ cout << feet << "\'-" << inches << '\"'; }
Distance operator + (Distance d2)
{
    int f = feet + d2.feet;
    float i = inches + d2.inches;
    if(i >= 12.0) //if total exceeds 12.0,
        { i -= 12.0; f++; }
    return Distance(f,i);
}
```

```
int main()
{
    Distance d1 = 2.5; //constructor converts
    Distance d2 = 1.25; //float feet to Distance
    Distance d3;
    cout << "\nd1 = "; d1.showdist();
    cout << "\nd2 = "; d2.showdist();
    d3 = d1 + 10.0;
    cout << "\nd3 = "; d3.showdist();
    // d3 = 10.0 + d1; //float + Distance: ERROR
    // cout << "\nd3 = "; d3.showdist();
    cout << endl;
    return 0;
}
```

d3 = Distance(10, 0) + d1; this will work but this is not good so As you may have guessed, a friend can help you out of this dilemma.

```
class Distance //English Distance class
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0) { }
Distance(float fltfeet) {
	feet = static_cast<int>(fltfeet);
	inches = 12*(fltfeet-feet);}
Distance(int ft, float in) { feet = ft; inches = in; }
void showdist()
{ cout << feet << "\'-" << inches << '\"'; }
friend Distance operator + (Distance, Distance)
};

Distance operator + (Distance d1, Distance d2) {
	int f = d1.feet + d2.feet;
	float i = d1.inches + d2.inches;
	if(i >= 12.0) //if inches exceeds 12.0,
		{ i -= 12.0; f++; }
	return Distance(f,i);
}
```

```
int main()
{
	Distance d1 = 2.5; //constructor converts
	Distance d2 = 1.25; //float-feet to Distance
	Distance d3;
	cout << "\nd1 = "; d1.showdist();
	cout << "\nd2 = "; d2.showdist();
	d3 = d1 + 10.0; //distance + float: OK
	cout << "\nd3 = "; d3.showdist();
	d3 = 10.0 + d1; //float + Distance: OK
	cout << "\nd3 = "; d3.showdist();
	cout << endl;
	return 0;
}
```

friend Classes

```
#include <iostream>
using namespace std;
class alpha
{
private:
int data1;
public:
alpha() : data1(99) { } //constructor
friend class beta; //beta is a friend class
};
class beta
{ //all member functions can
public: //access private alpha data
void func1(alpha a) { cout << “\ndata1=” << a.data1; }
void func2(alpha a) { cout << “\ndata1=” << a.data1; }
};
/////////////////////////////////////////////////////////////////
int main()
{
alpha a;
beta b;
b.func1(a);
b.func2(a);
cout << endl;
return 0;
}
```

In class alpha the entire class beta is proclaimed a friend. Now all the member functions of beta can access the private data of alpha

Overload a binary operator using non member function

```
class Complex
{
        private:
                float real;
                float imag;
        public:
                Complex(){}
                Complex(float r, float i)
                {
                        real = r;
                        imag = i;
                }
                void display()
                {
                        cout<<real<<"+i"<<imag;
                }
                friend Complex operator +(Complex &, Complex &);
};
Complex operator +(Complex &c1, Complex &c2)
{
        Complex temp;
        temp.real = c1.real + c2.real;
        temp.imag = c1.imag + c2.imag;
        return temp;
}
```

```
int main()
{
        Complex c1(3, 4);
        Complex c2(4, 6);
        Complex c3 = c1+c2;
        c3.display();
        return 0;
}
```

function is friendly for two classes.

```
#include<iostream>
using namespace std;
class B; //forward declaration.
class A
{
   int x;
   public:
       void setdata (int i)
         {
            x=i;
         }
   friend void max (A, B); //friend function.
} ;
class B
{
     int y;
     public:
          void setdata (int i)
            {
               y=i;
            }
     friend void max (A, B);
};
```

```
void max (A a, B b)
{
   if (a.x >= b.y)
         std:: cout<< a.x << std::endl;
   else
         std::cout<< b.y << std::endl;
}
  int main ()
{
   A a;
   B b;
    a. setdata (10);
    b. setdata (20);
    max (a, b);
    return 0;
}
```

We must know the following things before we start overloading these operators.

1) cout is an object of ostream class and cin is an object of istream class

2) These operators must be overloaded as a global function. And if we want to allow them to access private data members of the class, we must make them friend.

Why these operators must be overloaded as global?

In operator overloading, if an operator is overloaded as a member, then it must be a member of the object on the left side of the operator. For example, consider the statement "ob1 + ob2" (let ob1 and ob2 be objects of two different classes). To make this statement compile, we must overload '+' in a class of 'ob1' or make '+' a global function.

The operators '<<' and '>>' are called like 'cout << ob1' and 'cin >> ob1'. So if we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time. Therefore, these operators are overloaded as global functions with two parameters, cout and object of user-defined class.