National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2022, Lab Manual - 09

Course Code: CL-1004	**Course: Object Oriented Programming Lab**

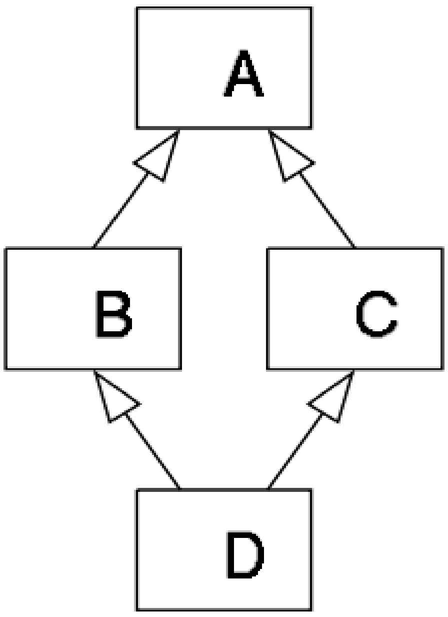
Lab # 09

Outline:

1. Diamond Problem in Hybrid Inheritance
2. Operator Overloading
3. Things to remember for Operator overloading
4. Lab Tasks

Diamond Problem in Hybrid Inheritance:

In case of hybrid inheritance, a Diamond problem may arise. The “dreaded diamond” refers to a class structure in which a particular class appears more than once in a class’s inheritance hierarchy.



Example of Diamond Problem:

```
#include<iostream>
using namespace std;
class A {
     public:
          int a;
};
class B : public A{
     public:
          int b;
};
class C : public A{
     public:
          int c;
};
class D : public B, public C{
     public:
          int d;
};
int main() {
     D obj;
     obj.a = 200; //will cause an error
}
```

How to Solve the Diamond Problem?

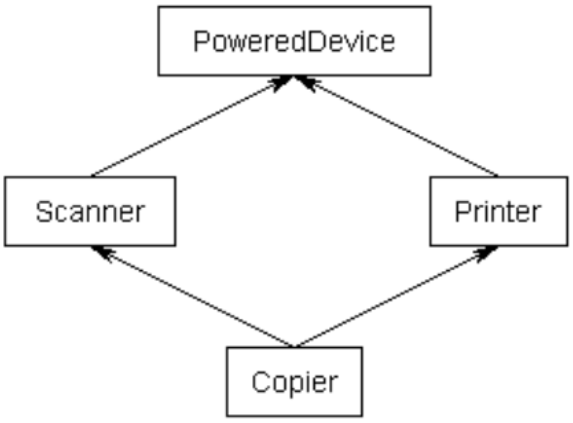
Answer: Virtual Base Classes

To share a base class, simply insert the "virtual" keyword in the inheritance list of the derived class. This creates what is called a **virtual base class**, which means there is only one base object. The base object is shared between all objects in the inheritance tree and it is only constructed once.

Solving the Diamond Problem:

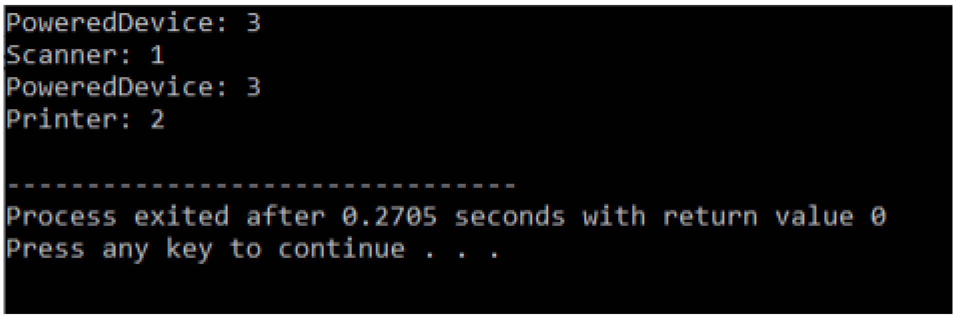
```
#include<iostream>
using namespace std;
class A {
     public:
          int a;
};
class B : virtual public A{ //adding the virtual keyword
     public:
          int b;
};
class C : virtual public A{ //adding the virtual keyword
     public:
          int c;
};
class D : public B, public C{
     public:
          int d;
};
int main() {
     D obj;
     obj.a = 200; //will no longer cause an error
}
```

Diamond Problem with Real Classes and Objects:



```
#include <iostream>
using namespace std;
class PoweredDevice {
      public:
            PoweredDevice(int power){
                  cout << "PoweredDevice: " << power << '\n';
            }
};
class Scanner: public PoweredDevice{
      public:
            Scanner(int scanner, int power) : PoweredDevice(power){
                  cout << "Scanner: " << scanner << '\n';
            }
};
class Printer: public PoweredDevice{
      public:
            Printer(int printer, int power) : PoweredDevice(power){
                  cout << "Printer: " << printer << '\n';
            }
};
class Copier: public Scanner, public Printer {
      public:
            Copier(int scanner, int printer, int power) : Scanner(scanner,
power), Printer(printer, power) { }
};
int main() {
      Copier copier(1, 2, 3);
      return 0;
 }
```

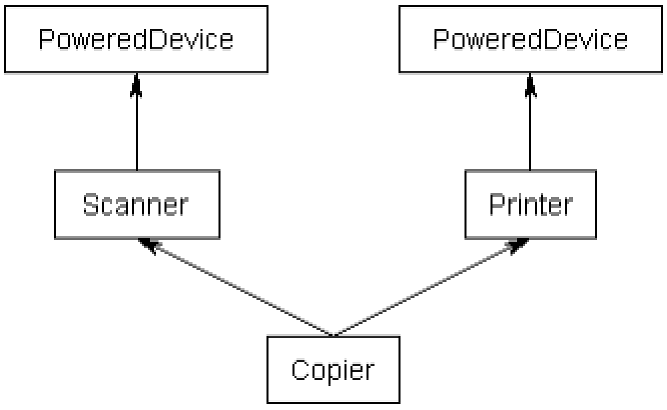
If you were to create a Copier class object, by default you would end up with two copies of the PoweredDevice class -- one from Printer, and one from Scanner. This has the following structure:



```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2

--------------------------------
Process exited after 0.2705 seconds with return value 0
Press any key to continue . . .
```

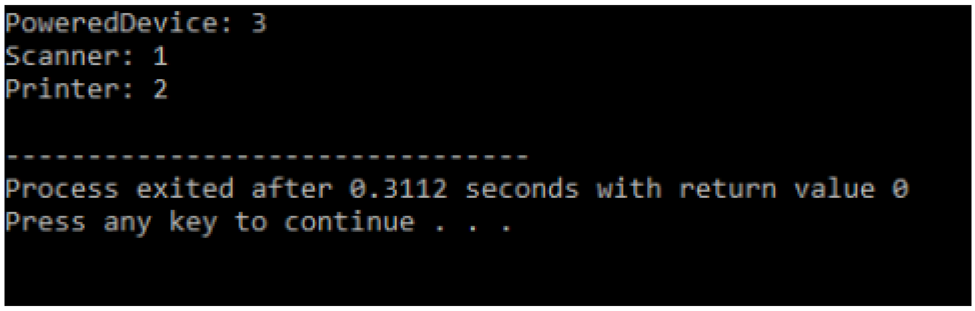
By using Virtual Base Classes:



```
#include <iostream>
using namespace std;
class PoweredDevice {
      public:
            PoweredDevice(int power){
                  cout << "PoweredDevice: " << power << '\n';
            }
};
class Scanner: virtual public PoweredDevice{ // note: PoweredDevice is now a virtual base class
      public:
            Scanner(int scanner, int power) : PoweredDevice(power){   // this line is required to create Scanner objects, but ignored in this case
                  cout << "Scanner: " << scanner << '\n';
            }
};
class Printer: virtual public PoweredDevice{ // note: PoweredDevice is now a virtual base class
```

```
      public:
            Printer(int  printer,  int  power)  :  PoweredDevice(power){  //  this
line is required to create Printer objects, but ignored in this case
                  cout << "Printer: " << printer << '\n';
            }
};
class Copier: public Scanner, public Printer {
      public:
            Copier(int scanner, int printer, int power) : PoweredDevice(power),
Scanner(scanner,   power),   Printer(printer,   power)   //   PoweredDevice   is
constructed here at PoweredDevice(power)
            { }
};
int main() {
      Copier copier(1, 2, 3);
      return 0;
 }
```

Now, when you create a Copier class object, you will get only one copy of PoweredDevice per Copier that will be shared by both Scanner and Printer. However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsible for creating it?
The answer, as it turns out, is Copier. The Copier constructor is responsible for creating PoweredDevice. Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly.



```
PoweredDevice: 3
Scanner: 1
Printer: 2

--------------------------------
Process exited after 0.3112 seconds with return value 0
Press any key to continue . . .
```

Operator Overloading:

In C++, the purpose of operator overloading for an operator is to specify more than one meaning in one scope. It gives special meaning of an operator for a user-defined data type.

You can redefine most of the C++ operators with the help of operator overloading. To perform multiple operations using one operator, you can also use operator overloading.

Syntax for operator overloading:

To overload a C++ operator, you should define a special function inside the Class as follows:

```
class class_name
{
    ... .. ...
    public
       return_type operator symbol (argument(s))
       {
           ... .. ...
       }
    ... .. ...
};
```

Here is an explanation for the above syntax:

- The return_type is the return type for the function.
- Next, you mention the operator keyword.
- The symbol denotes the operator symbol to be overloaded. For example, +, -, <, ++.
- The argument(s) can be passed to the operator function in the same way as functions.

Example :

```
#include <iostream>
using namespace std;
class TestClass {
private:
      int count;
public:
      TestClass()
      {
      count=5;
      }
      void operator --() {
            count = count - 3;
```

```
        }
        void Display() {
                cout << "Count: " << count; }
};
int main() {
        TestClass tc;
        --tc;
        tc.Display();
        return 0;}
```

Output:

```
Count: 2
```

C++ Operators that cannot be Overloaded

There are four C++ operators that can't be overloaded.

They include:

- :: Scope resolution operator
- ?: ternary operator.
- . member selection operator
- .* member selection through a pointer to function operator.

C++ Operators that can be Overloaded

- Following is the list of operators which can be overloaded –

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=

|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Unary Operators Overloading:

The unary operators operate on a single operand and following are the examples of Unary operators –

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

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

class Distance {
   private:
      int feet;              // 0 to infinite
      int inches;            // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }

      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};

int main() {
   Distance D1(11, 10), D2(-5, 11);

   -D1;                     // apply negation
   D1.displayDistance();    // display D1

   -D2;                     // apply negation
   D2.displayDistance();    // display D2

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
F: -11 I:-10
F: 5 I:-11
```

Hope above example makes your concept clear and you can apply similar concept to overload Logical Not Operators (!).

The increment (++) and decrement (--) operators are two important unary operators available in C++. Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way, you can overload operator (--).

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

class Time {
   private:
      int hours;             // 0 to 23
      int minutes;           // 0 to 59

   public:
      // required constructors
      Time() {
         hours = 0;
         minutes = 0;
      }
      Time(int h, int m) {
         hours = h;
         minutes = m;
      }

      // method to display time
      void displayTime() {
         cout << "H: " << hours << " M:" << minutes <<endl;
      }

      // overloaded prefix ++ operator
      Time operator++ () {
         ++minutes;          // increment this object
         if(minutes >= 60) {
            ++hours;
            minutes -= 60;
         }
         return Time(hours, minutes);
      }

      // overloaded postfix ++ operator
      Time operator++( int ) {

         // save the original value
         Time T(hours, minutes);

         // increment this object
         ++minutes;

         if(minutes >= 60) {
            ++hours;
            minutes -= 60;
         }

         // return old original value
         return T;
      }
};

int main() {
   Time T1(11, 59), T2(10,40);

   ++T1;                    // increment T1
   T1.displayTime();        // display T1
   ++T1;                    // increment T1 again
   T1.displayTime();        // display T1

   T2++;                    // increment T2
   T2.displayTime();        // display T2
   T2++;                    // increment T2 again
   T2.displayTime();        // display T2
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42
```

Binary Operators Overloading:

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

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

class Box {
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box

   public:

   double getVolume(void) {
      return length * breadth * height;
   }

   void setLength( double len ) {
      length = len;
   }

   void setBreadth( double bre ) {
      breadth = bre;
   }

   void setHeight( double hei ) {
      height = hei;
   }

   // Overload + operator to add two Box objects.
   Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
};

// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here
   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Relational Operators Overloading:

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.

You can overload any of these operators, which can be used to compare the objects of a class.

Following example explains how a < operator can be overloaded and similar way you can overload other relational operators.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }


      // overloaded < operator
      bool operator <(const Distance& d) {
         if(feet < d.feet) {
            return true;
         }
         if(feet == d.feet && inches < d.inches) {
            return true;
         }

         return false;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11);

   if( D1 < D2 ) {
      cout << "D1 is less than D2 " << endl;
   } else {
      cout << "D2 is less than D1 " << endl;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
D2 is less than D1
```

Assignment Operator Overloading:

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.

Following example explains how an assignment operator can be overloaded.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      void operator = (const Distance &D ) {
         feet = D.feet;
         inches = D.inches;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // use assignment operator
   D1 = D2;
   cout << "First Distance :";
   D1.displayDistance();
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

Function Call() Operator Overloading:

The function call operator () can be overloaded for objects of class type. When you overload ( ), you are not creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Following example explains how a function call operator () can be overloaded.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // overload function call
      Distance operator()(int a, int b, int c) {
         Distance D;

         // just put random calculation
         D.feet = a + c + 10;
         D.inches = b + c + 100 ;
         return D;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches << endl;
      }
};
```

```
int main() {
   Distance D1(11, 10), D2;

   cout << "First Distance : ";
   D1.displayDistance();

   D2 = D1(10, 10, 10); // invoke operator()
   cout << "Second Distance :";
   D2.displayDistance();

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
First Distance : F: 11 I:10
Second Distance :F: 30 I:120
```

Things to remember for Operator Overloading:

Here are rules for Operator Overloading:

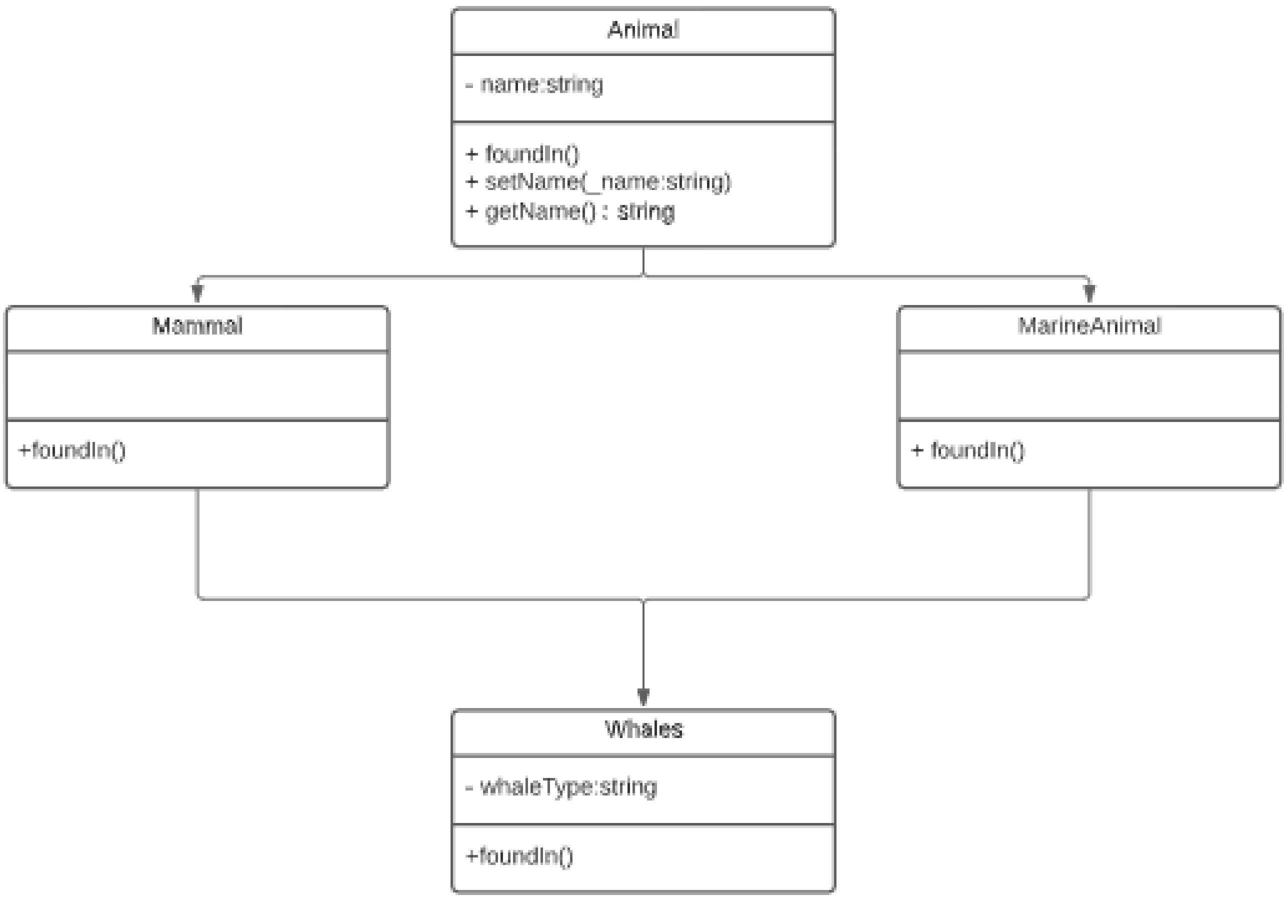
- At least one of the operands in overloaded operators must be user-defined, which means we cannot overload the minus operator to work with one integer and one double. However, you could overload the minus operator to work with an integer and a mystring.
- We can only overload the operators that exist and cannot create new operators or rename existing operators.
- You can make the operator overloading function a friend function if it needs to access the private and protected class members.
- Some operators cannot be overloaded using a friend function. However, such operators can be overloaded using member function. Eg. Assignment overloading operator = and function call() operator etc.

LAB TASKS:

Task - 00:

Follow the instructions during class.

Task - 01:



Create a class, as depicted in the diagram. Make sure to use virtual base classes for this task.
Make an object **whaleBaleen**. Then set the **whaleType to "Baleen", and name to "whale"**

Write the following in the function descriptions for **foundIn()**

- **Animal::foundIn()** – Prints: An animal can be found in many places
- **Mammal::foundIn()** – Prints: A mammal can be found in water or on land
- **MarineAnimal::foundIn()** – Prints: A marine animal can only be found in bodies of water
- **Whales::foundIn()** – Prints: A <whaleType> <name> can only be found in the ocean.

Task - 02:

Create a class called Distance (similar to the previous Distance example). It should have two private variables called **feet** and **inches.**

- Overload the post-increment operator such that it increments inches by 1.
- Overload the post-decrement operator such that it decrements inches by 1.

You should make sure that inches do not exceed 11, as 12 inches = 1 foot. **You should also make sure the the value for inches does not become negative.**

For example:

- If you increment 5’11” by one, it should become 6’0”
- If you decrement 5’0” by one, it should become 4’11”

Task - 03:

Create a class called Box with the following attributes and functions:

- Number of sides
- Area
- A display function that displays the number of sides and area of the box

Write a program to overload the binary operators **+**, **>** and **<** for the class. Create three objects named **box1**, **box2** and **resultantBox**.

- Overload **+** to return a Box type object which contain the sum of sides and areas for the two box objects **box1** and **box2**. Store the result in **resultantBox** and then display the result
- Overload **>** to assign value of its **left operand** to its **right operand**.
- Overload **<** to assign value of its **right operand** to its **left operand**.