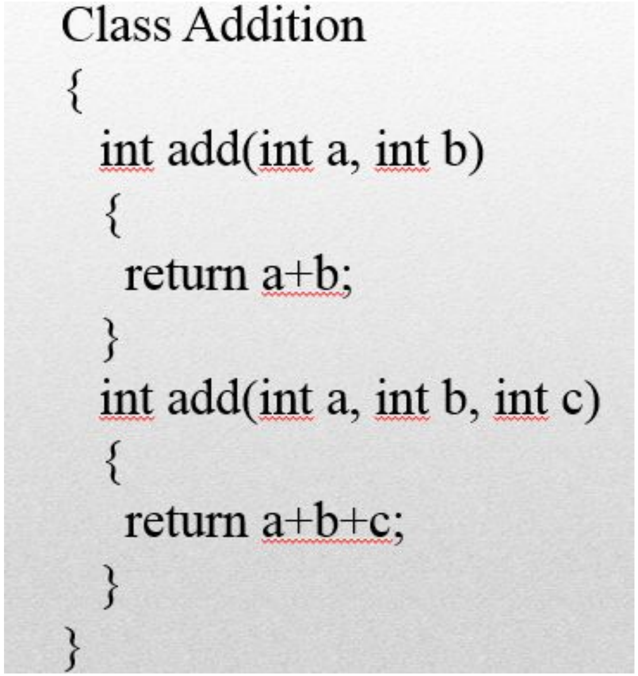
Week 9: Operator Overloading,Diamond Problem, Friend Functions

By, Mahrukh Khan

Overloading

- Function Overloading
- Constructor Overloading
- Operator Overloading

Function Overloading



```
Class Addition
{
  int add(int a, int b)
  {
   return a+b;
  }
  int add(int a, int b, int c)
  {
   return a+b+c;
  }
}
```

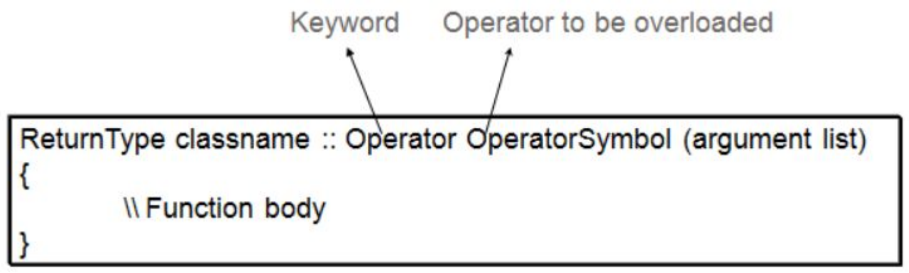
Constructor Overloading

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments.This concept is known as Constructor Overloading and is quite similar to function overloading.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

Operator Overloading

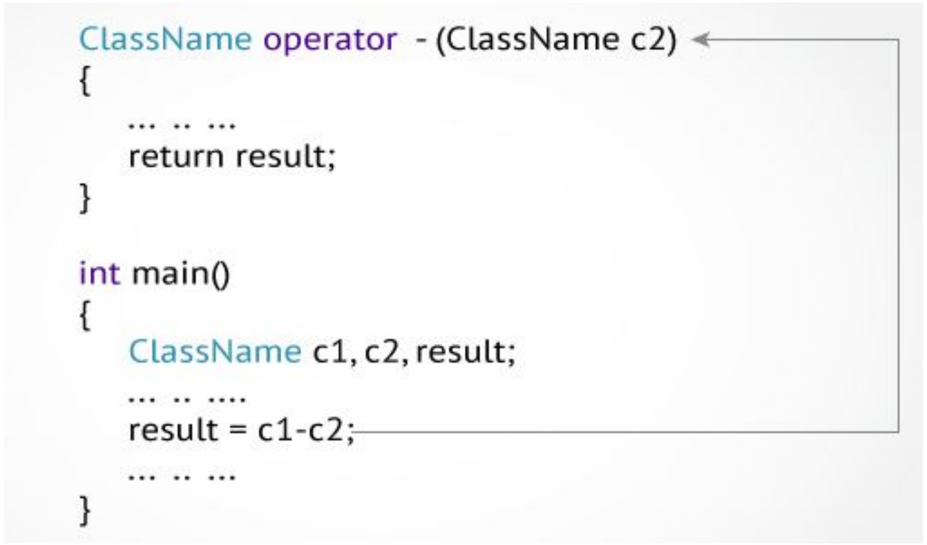
- An operator is a symbol that tells the compiler to perform specific task. Every operator have their own functionality to work with built-in data types.Class is user-defined data type and compiler doesn't understand, how to use operators with user-defined data types.
- You can redefine or overload most of the built-in operators available in C++
- An operator can be overloaded by defining a function to it. The function for operator is declared by using the **operator** keyword followed by the operator. Like any other function, an overloaded operator has a return type and a parameter list.



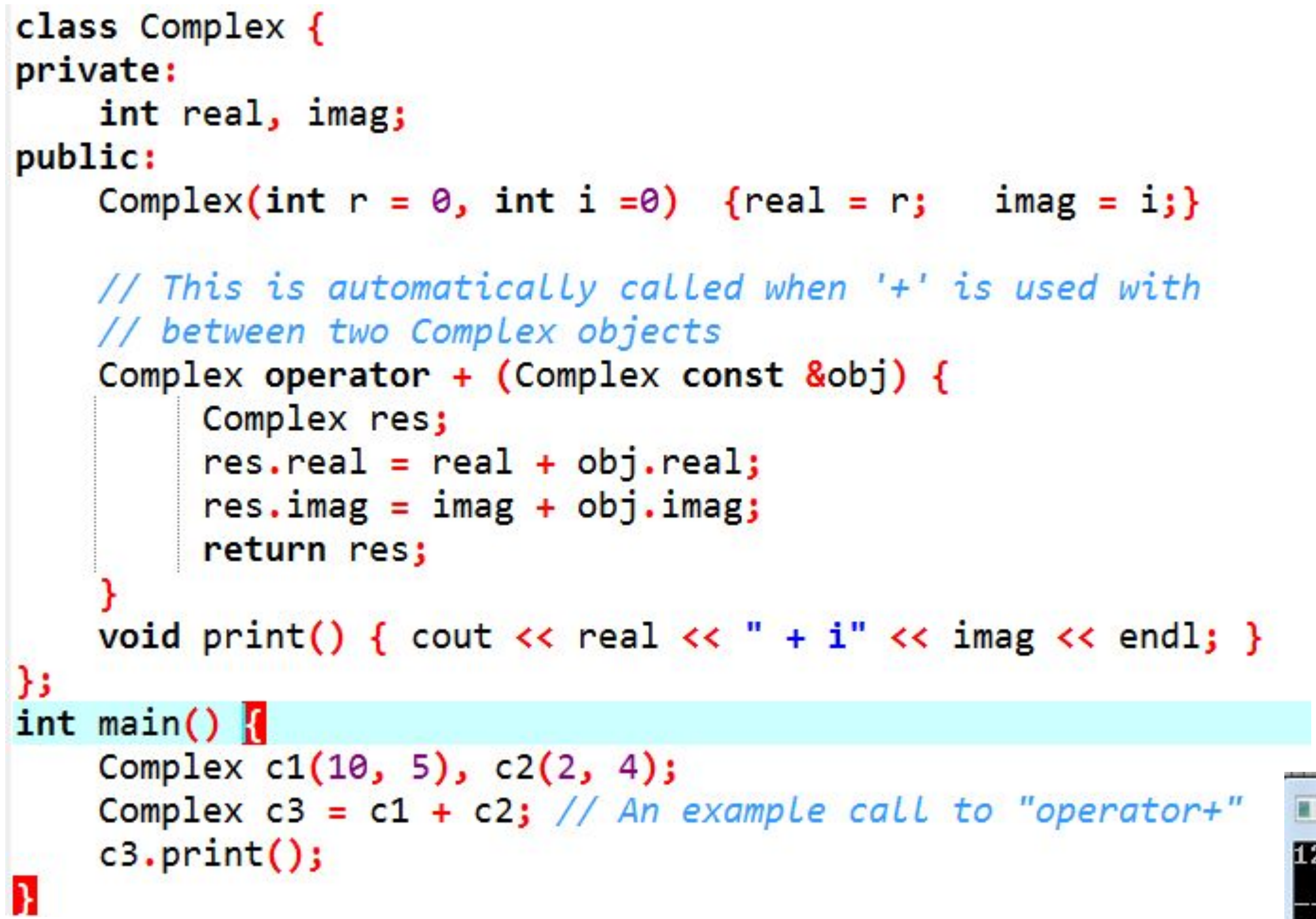
```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Syntax

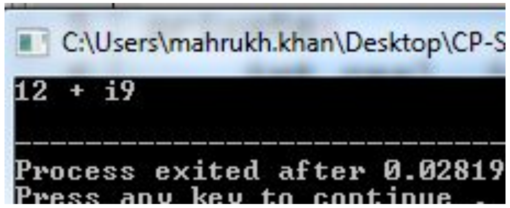
```
class className
{
    ... .. ...
    public
       returnType operator symbol (arguments)
       {
           ... .. ...
       }
    ... .. ...
};
```



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

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



```cpp
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();
}
```

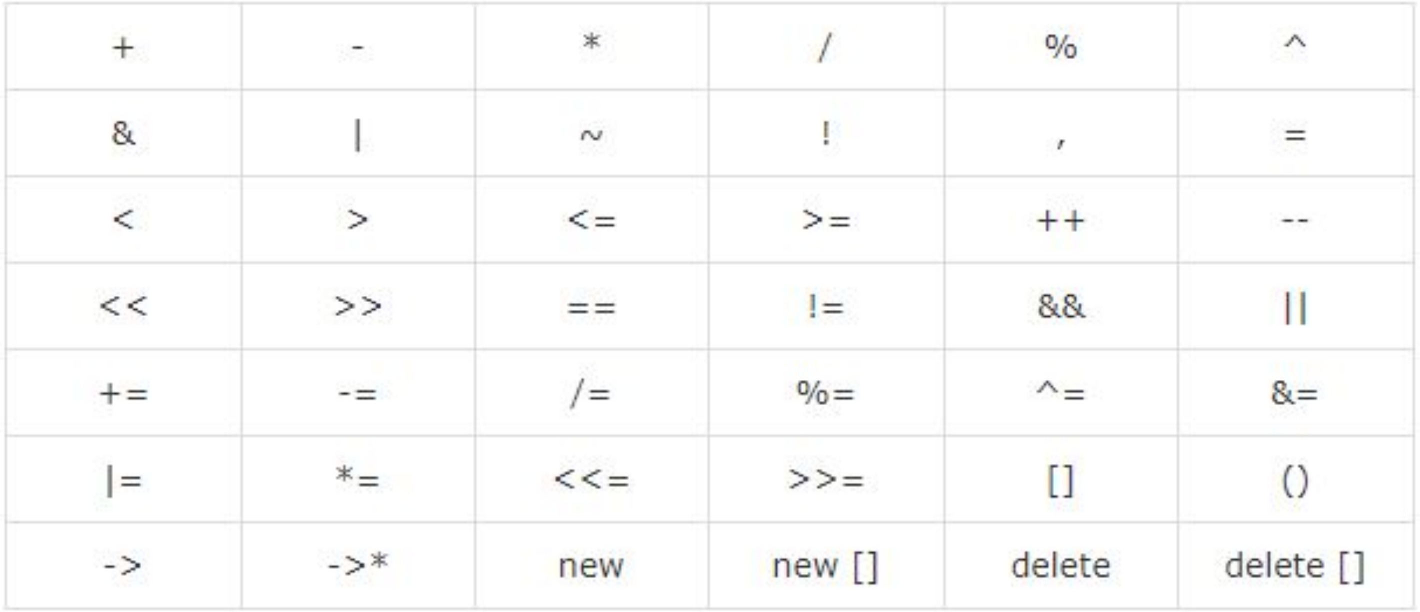


```
C:\Users\mahrukh.khan\Desktop\CP-S
12 + i9

--------------------------------
Process exited after 0.02819
Press any key to continue .
```

Operator Overloading

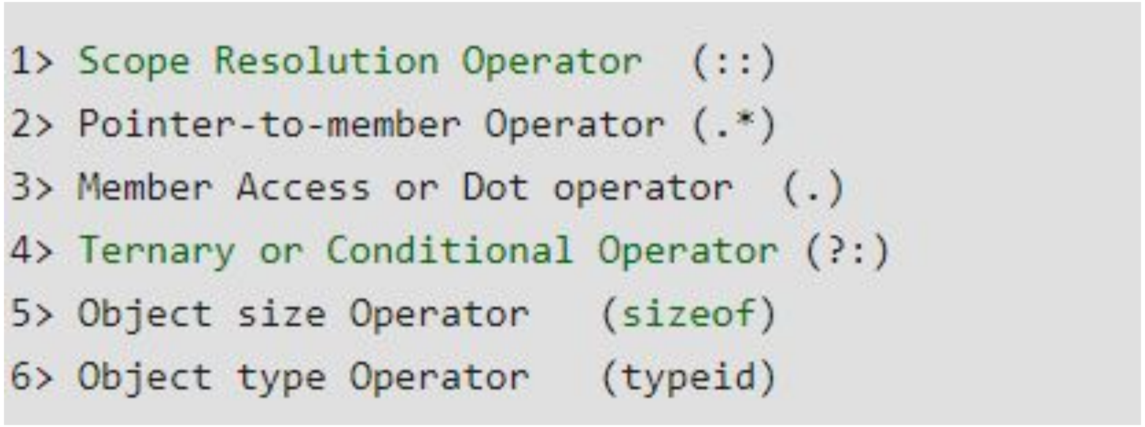
- When you call an overloaded function or operator, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.
- 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.).
- 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. 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).

Overloadable Operators



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

Non-Overloadable Operators



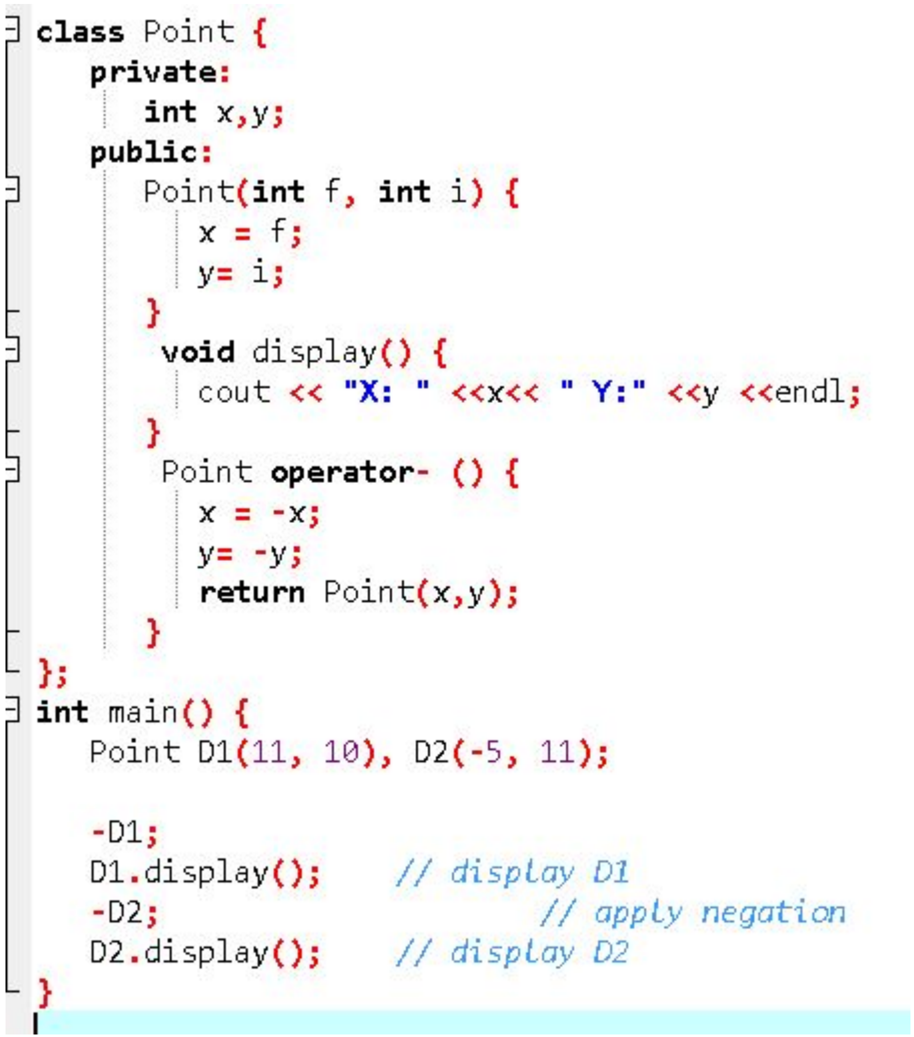
```
1> Scope Resolution Operator  (::)
2> Pointer-to-member Operator (.*)
3> Member Access or Dot operator  (.)
4> Ternary or Conditional Operator (?:)
5> Object size Operator   (sizeof)
6> Object type Operator   (typeid)
```

Overloadable Operators

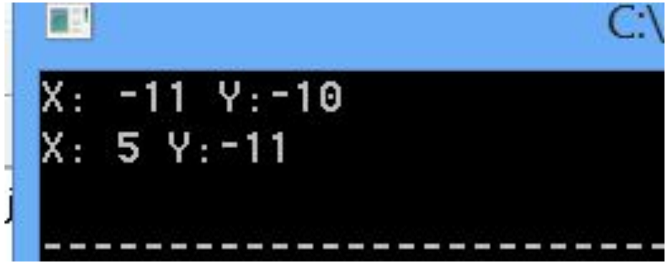
- Unary
- Binary
- Relational
- Input/Output
- Increment/Decrement
- Assignment Operators
- Function Call() Operator Overloading
- Subscripting [ ] Operator Overloading
- Class member Access Operator Overloading

Unary

- 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--.



```cpp
class Point {
    private:
        int x,y;
    public:
        Point(int f, int i) {
            x = f;
            y= i;
        }
         void display() {
            cout << "X: " <<x<< " Y:" <<y <<endl;
        }
         Point operator- () {
            x = -x;
            y= -y;
            return Point(x,y);
        }
};
int main() {
    Point D1(11, 10), D2(-5, 11);

    -D1;
    D1.display();     // display D1
    -D2;                      // apply negation
    D2.display();     // display D2
}
```



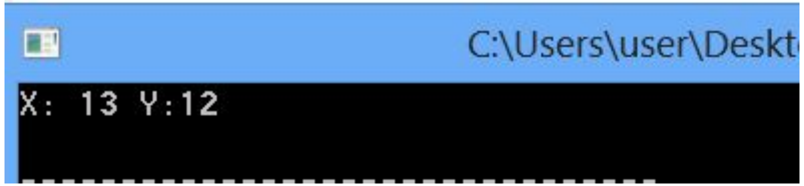
```
X: -11 Y:-10
X: 5 Y:-11

--------------------------
```

Binary

- 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.

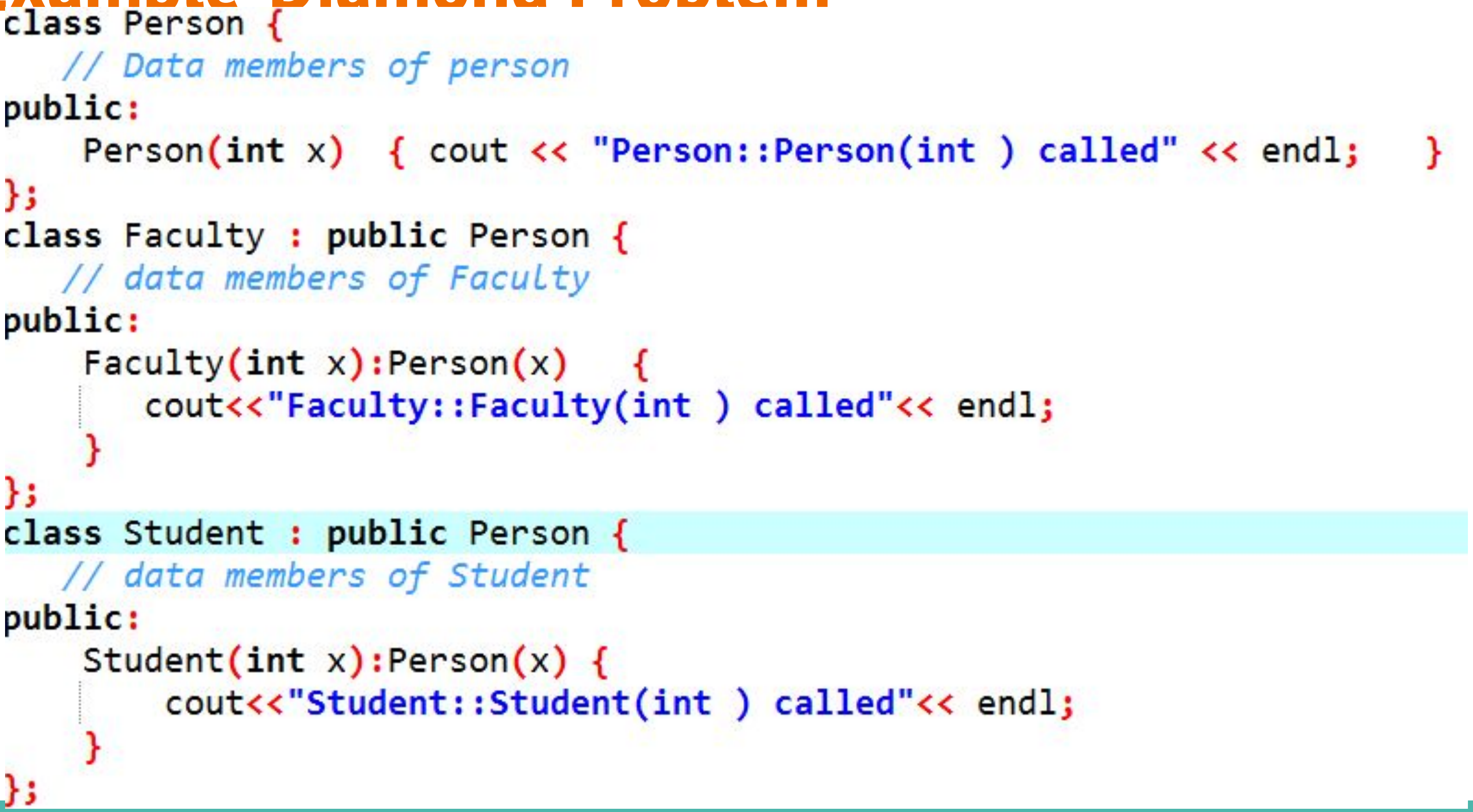
```
class Point {
   private:
      int x,y;
   public:
    Point(){
       }
      Point(int f, int i) {
         x = f;
         y= i;
      }
       void display() {
         cout << "X: " <<x<< " Y:" <<y <<endl;
      }
       Point operator- () {
         x = -x;
         y= -y;
         return Point(x,y);
      }
       Point operator+(const Point& b) {
      Point p;
      p.x = this->x + b.x;
      p.y = this->y + b.y;
      return p;
   }
};
```

```
int main() {
   Point D1(11, 10), D2(2,2);
   Point D3;
   D3=D1+D2;
   D3.display();
}
```

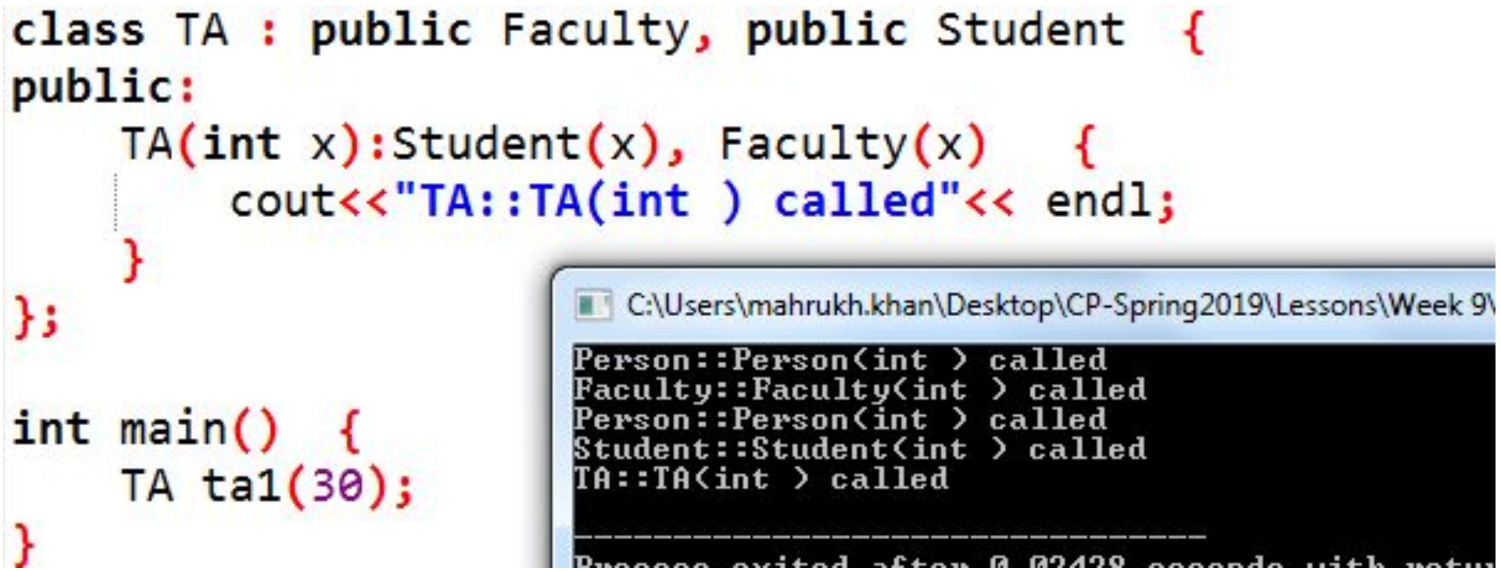


```
X: 13 Y:12
```

Example-Diamond Problem



```
class Person {
   // Data members of person
public:
    Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
};
class Faculty : public Person {
   // data members of Faculty
public:
    Faculty(int x):Person(x)   {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
class Student : public Person {
   // data members of Student
public:
    Student(int x):Person(x) {
         cout<<"Student::Student(int ) called"<< endl;
    }
};
```

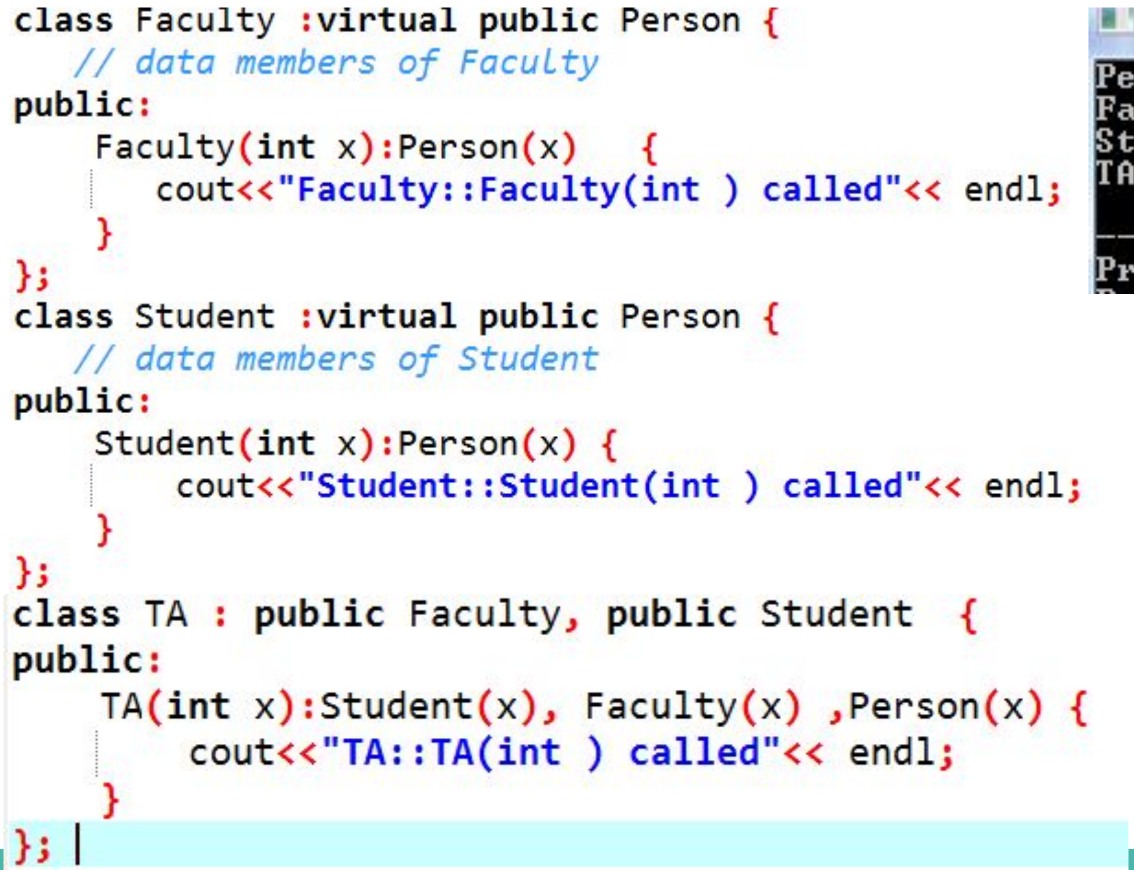


```cpp
class TA : public Faculty, public Student  {
public:
    TA(int x):Student(x), Faculty(x)   {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main()  {
    TA ta1(30);
}
```

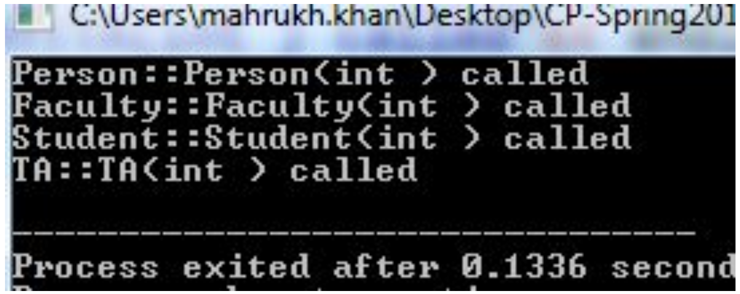
C:\Users\mahrukh.khan\Desktop\CP-Spring2019\Lessons\Week 9\

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called

--------------------------------
Process exited after 0.03428 seconds with retu
```

Solution-virtual inheritance



```cpp
class Faculty :virtual public Person {
   // data members of Faculty
public:
    Faculty(int x):Person(x)    {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
class Student :virtual public Person {
   // data members of Student
public:
    Student(int x):Person(x) {
         cout<<"Student::Student(int ) called"<< endl;
    }
};
class TA : public Faculty, public Student  {
public:
    TA(int x):Student(x), Faculty(x) ,Person(x) {
         cout<<"TA::TA(int ) called"<< endl;
    }
};
```



```
C:\Users\mahrukh.khan\Desktop\CP-Spring201
Person::Person(int ) called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called

--------------------------------
Process exited after 0.1336 second
```

Virtual Inheritance

- Virtual inheritance *solves* the classic “Diamond Problem”. It ensures that the child class gets only a single instance of the common base class.
- In other words, the **TA** class will have only **one** instance of the **Person** class. The **Student** and **Faculty** classes share this instance.

Friend Functions

A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. It works as bridge between classes.

- Friend function must be declared with **friend** keyword.
- Friend function must be declare in all the classes from which we need to access private or protected members.
- Friend function will be defined outside the class without specifying the class name.
- Friend function will be invoked like normal function, without any object.