Lecture 8
OOP

Khola Naseem
khola.naseem@uet.edu.pk

Dynamic memory allocation:

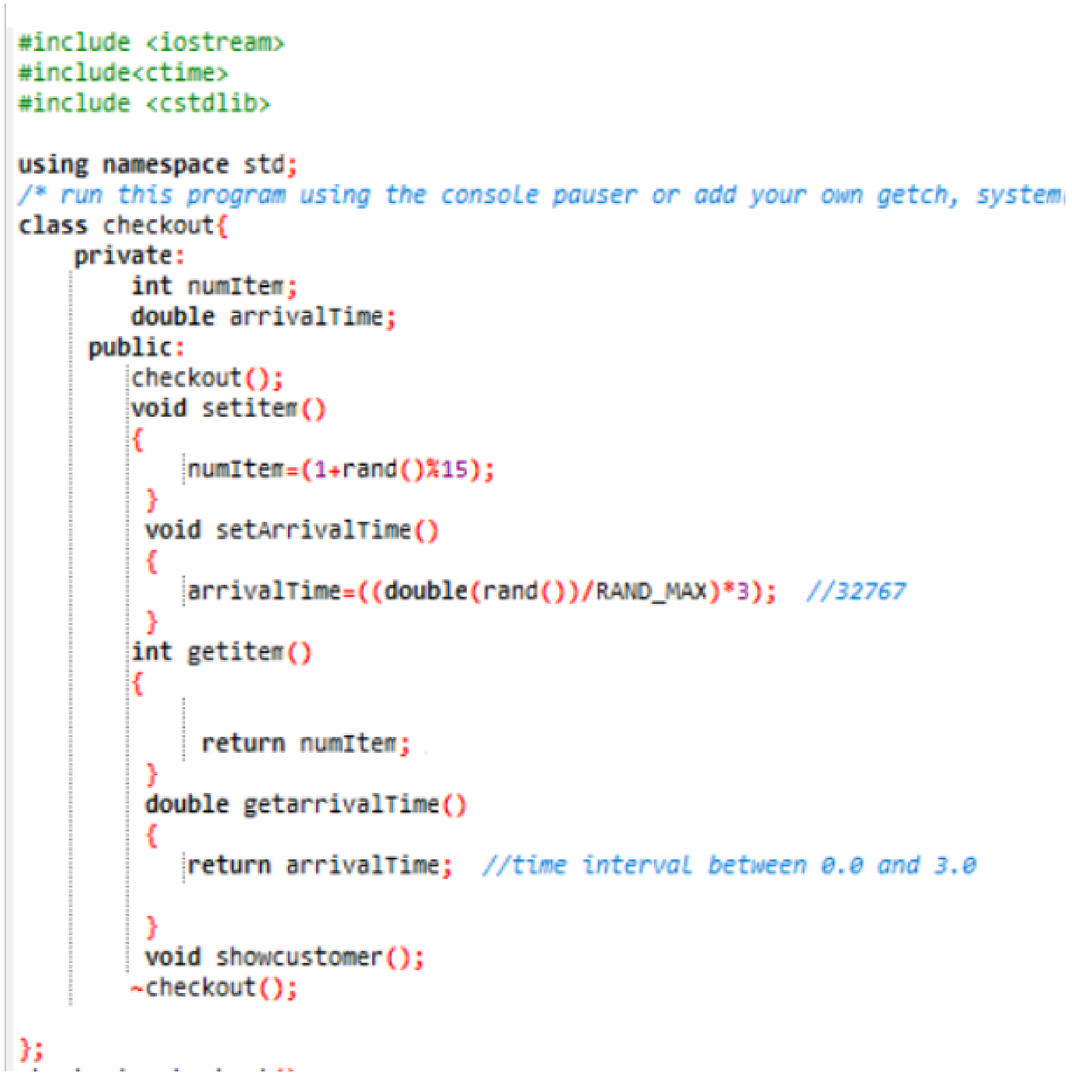
- **creation of Objects**
- You’re creating a program designed to track the flow of customers at a large outlet chain’s self-checkout registers. You don’t know in advance the number of transactions taking. However, by using dynamic allocation, the program can create an object as each checkout transaction occurs for tracking customer use of the checkout process. This object might consist of the number of items checked out and the arrival time of each customer at the register.

Dynamic memory allocation:

- **Objects**
- you’re creating a program designed to track the flow of customers at a large outlet chain’s self-checkout registers. You don’t know in advance the number of transactions taking place. However, by using dynamic allocation, the program can create an object as each checkout transaction occurs for tracking customer use of the checkout process. This object might consist of the number of items checked out and the arrival time of each customer at the register
- Library files
  - #include<ctime>
  - #include <cstdlib>

Dynamic memory allocation:

➢ **Objects**

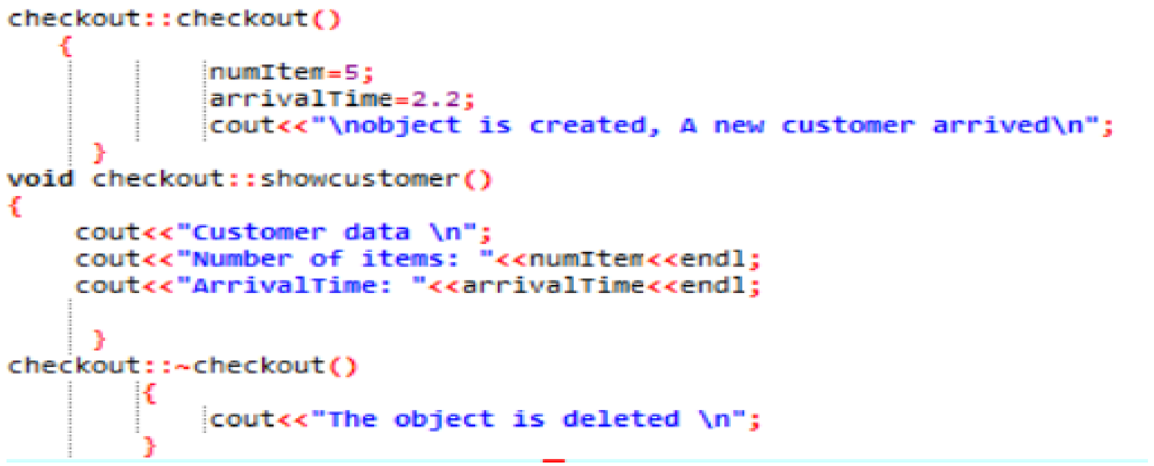


```
#include <iostream>
#include<ctime>
#include <cstdlib>

using namespace std;
/* run this program using the console pauser or add your own getch, system
class checkout{
    private:
        int numItem;
        double arrivalTime;
     public:
        checkout();
        void setitem()
        {
            numItem=(1+rand()%15);
         }
         void setArrivalTime()
         {
            arrivalTime=((double(rand())/RAND_MAX)*3);   //32767
         }
        int getitem()
        {

             return numItem;
         }
         double getarrivalTime()
         {
            return arrivalTime;   //time interval between 0.0 and 3.0

         }
         void showcustomer();
        ~checkout();

};
```

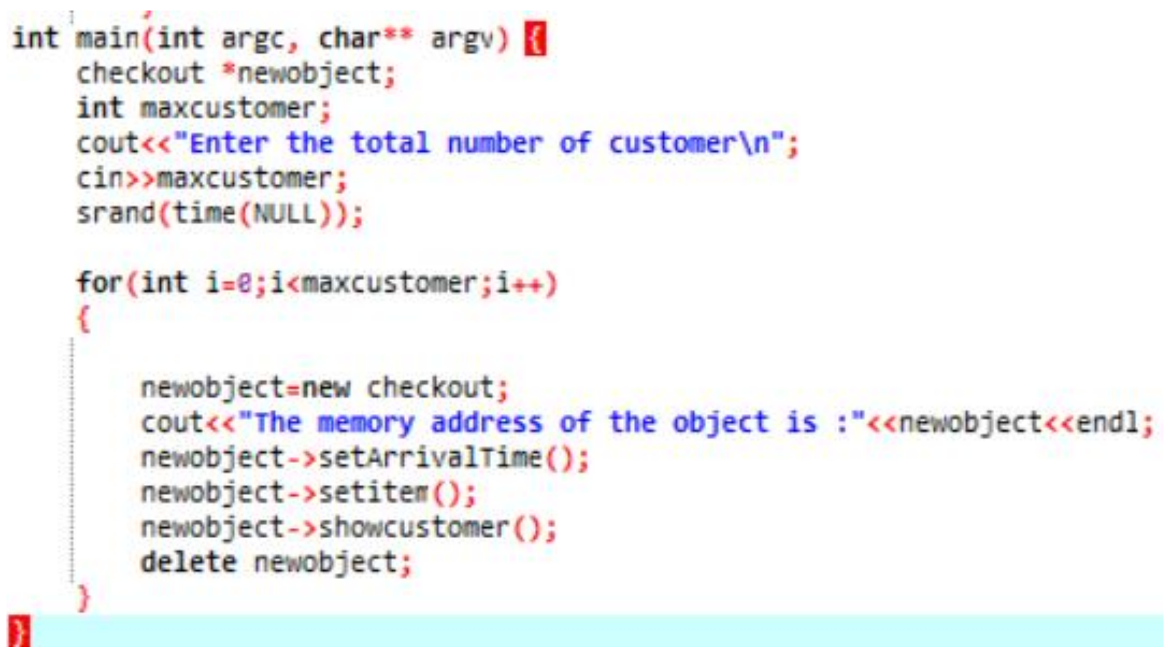
Credit: Khola Naseem

Dynamic memory allocation:

- **Objects**



```
checkout::checkout()
   {
            numItem=5;
            arrivalTime=2.2;
            cout<<"\nobject is created, A new customer arrived\n";
   }
void checkout::showcustomer()
{
    cout<<"Customer data \n";
    cout<<"Number of items: "<<numItem<<endl;
    cout<<"ArrivalTime: "<<arrivalTime<<endl;

   }
checkout::~checkout()
        {
            cout<<"The object is deleted \n";
        }
```



```
int main(int argc, char** argv) {
    checkout *newobject;
    int maxcustomer;
    cout<<"Enter the total number of customer\n";
    cin>>maxcustomer;
    srand(time(NULL));

    for(int i=0;i<maxcustomer;i++)
    {

        newobject=new checkout;
        cout<<"The memory address of the object is :"<<newobject<<endl;
        newobject->setArrivalTime();
        newobject->setitem();
        newobject->showcustomer();
        delete newobject;
    }
}
```

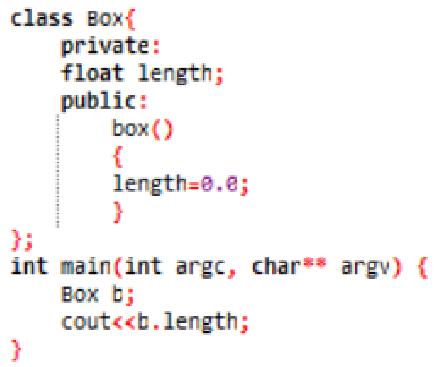
Credit: Khola Naseem

Dynamic memory allocation:

- **Objects**
- Deleting dynamically created objects when their usefulness ends is crucial.
- Otherwise, as new objects are created, the computer starts to “eat up” available memory space, especially if the same pointer is used in creating a new object before the old object is deleted.
- Reason
- This condition is referred to as memory leak
- In worst case as available memory is lost because of a memory leak, system operation can slow down, applications can fail, and the computer can crash.

Friend function:

- Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.

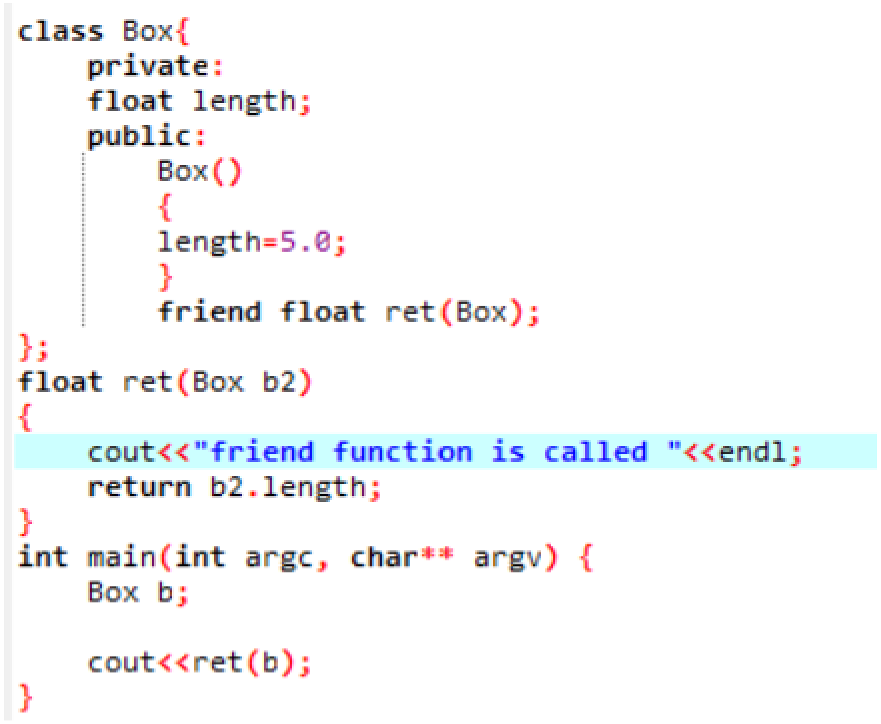


```
class Box{
    private:
    float length;
    public:
        box()
        {
        length=0.0;
        }
};
int main(int argc, char** argv) {
    Box b;
    cout<<b.length;
}
```

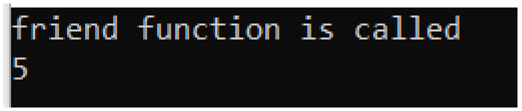
- However, there is a feature in C++ called friend functions that break this rule and allow us to access members from outside the class.
- Similarly, there is a friend class as well
- They are defined globally outside the class scope.
- How? The class maintains an approved list of nonmember methods that are granted the same privileges as its member methods. The nonmember methods in the list are called **friend functions,**

Friend function:

- .



```
class Box{
    private:
    float length;
    public:
        Box()
        {
        length=5.0;
        }
        friend float ret(Box);
};
float ret(Box b2)
{
    cout<<"friend function is called "<<endl;
    return b2.length;
}
int main(int argc, char** argv) {
    Box b;

    cout<<ret(b);
}
```

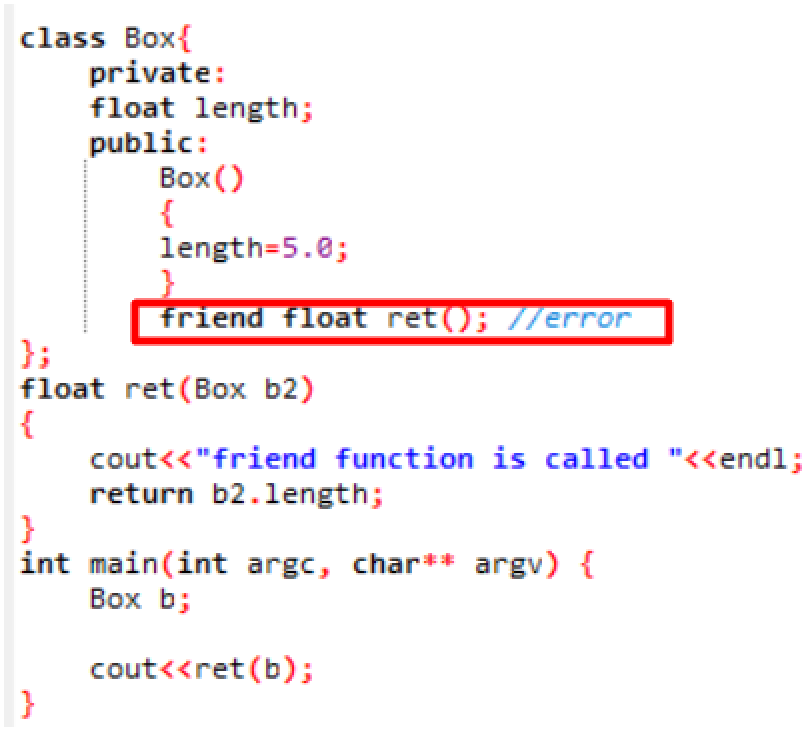


```
friend function is called
5
```

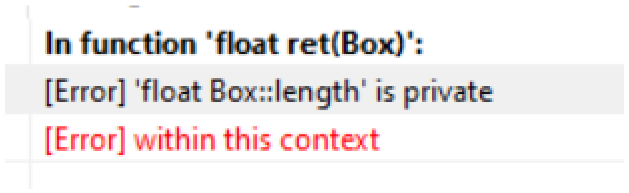
Credit: Khola Naseem

Friend function:

- .



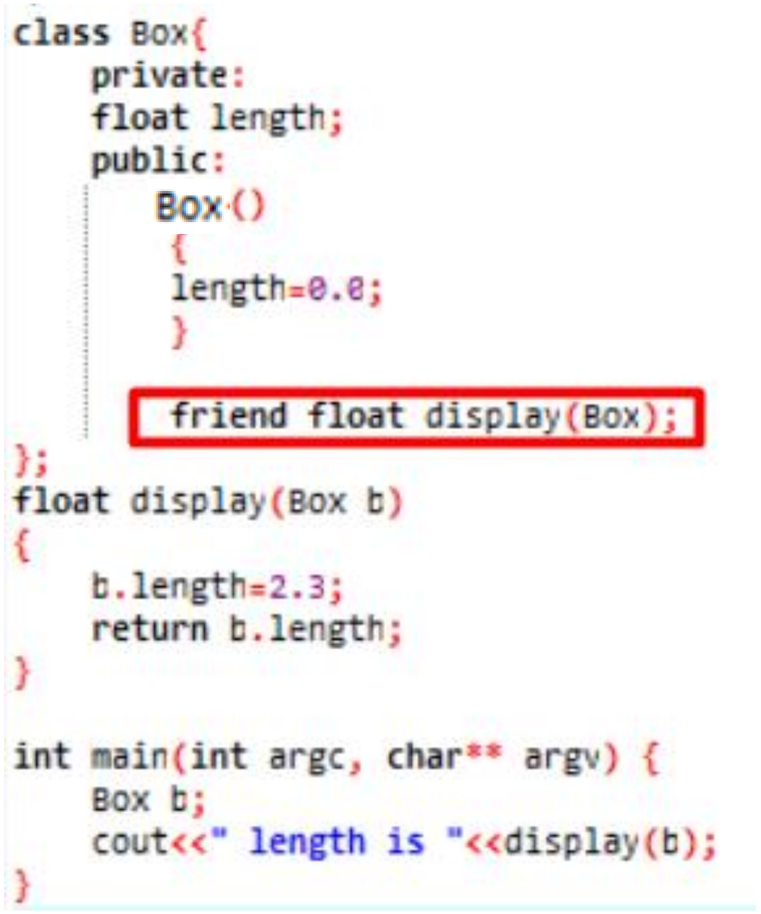
```
class Box{
    private:
    float length;
    public:
        Box()
        {
        length=5.0;
        }
        friend float ret(); //error
};
float ret(Box b2)
{
    cout<<"friend function is called "<<endl;
    return b2.length;
}
int main(int argc, char** argv) {
    Box b;

    cout<<ret(b);
}
```



```
In function 'float ret(Box)':
[Error] 'float Box::length' is private
[Error] within this context
```

Friend function:

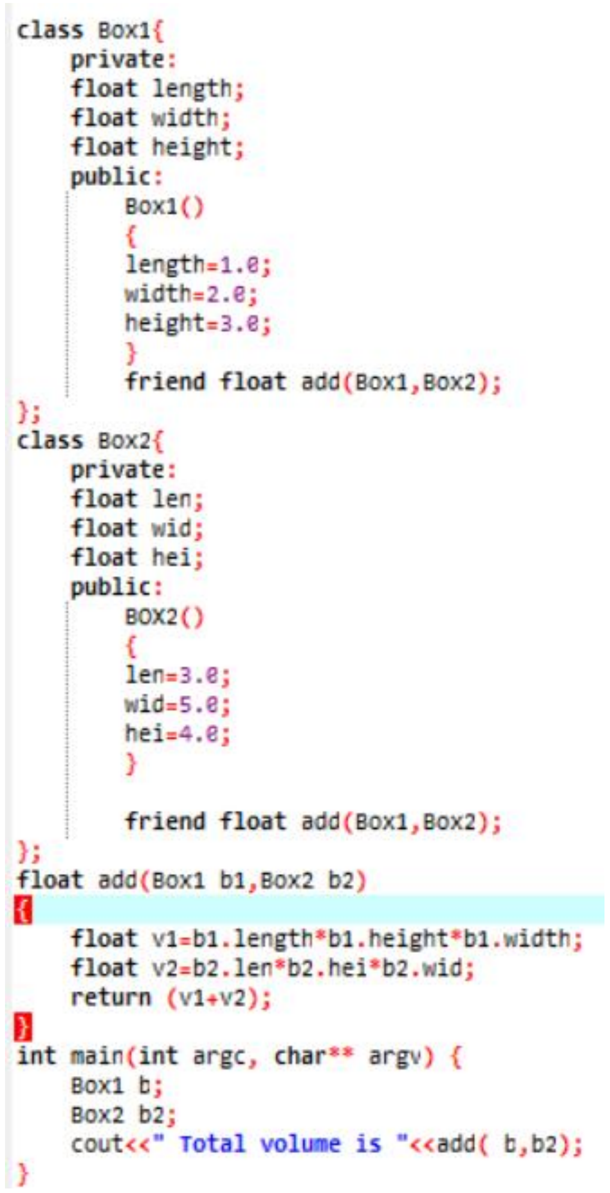
- From a coding standpoint, the friends list is simply a series of method prototype declarations preceded with the keyword friend and included in the class declaration section.



```
class Box{
    private:
    float length;
    public:
        Box()
        {
        length=0.0;
        }

        friend float display(Box);
};
float display(Box b)
{
    b.length=2.3;
    return b.length;
}

int main(int argc, char** argv) {
    Box b;
    cout<<" length is "<<display(b);
}
```

Friend function:

- Example:

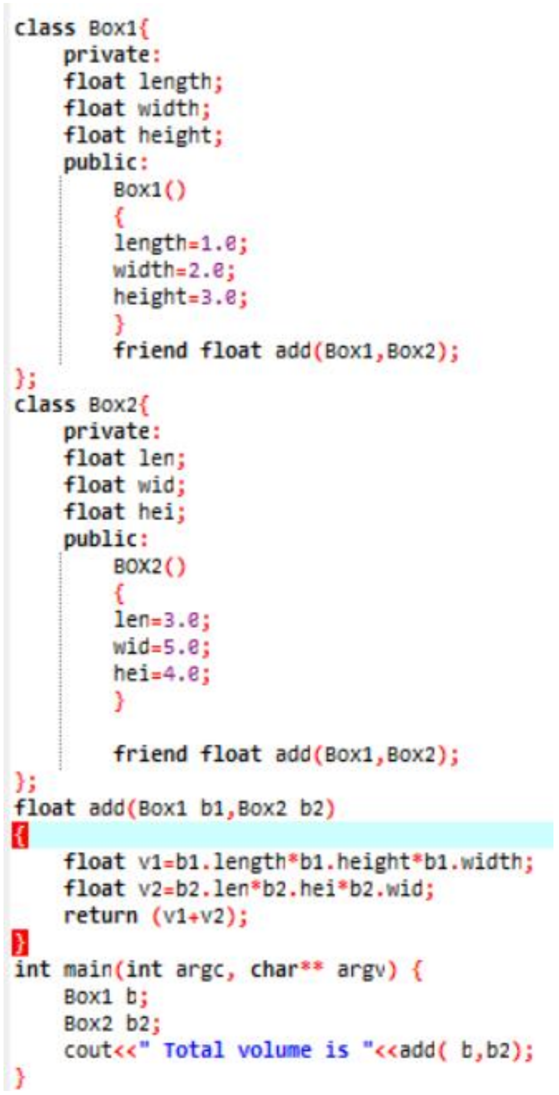


```
class Box1{
    private:
    float length;
    float width;
    float height;
    public:
        Box1()
        {
        length=1.0;
        width=2.0;
        height=3.0;
        }
        friend float add(Box1,Box2);
};
class Box2{
    private:
    float len;
    float wid;
    float hei;
    public:
        BOX2()
        {
        len=3.0;
        wid=5.0;
        hei=4.0;
        }

        friend float add(Box1,Box2);
};
float add(Box1 b1,Box2 b2)
{
    float v1=b1.length*b1.height*b1.width;
    float v2=b2.len*b2.hei*b2.wid;
    return (v1+v2);
}
int main(int argc, char** argv) {
    Box1 b;
    Box2 b2;
    cout<<" Total volume is "<<add( b,b2);
}
```

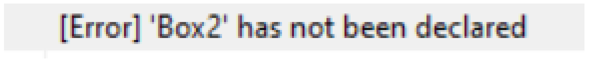
Credit: Khola Naseem

Friend function:

➢ Example:



```
class Box1{
    private:
    float length;
    float width;
    float height;
    public:
        Box1()
        {
        length=1.0;
        width=2.0;
        height=3.0;
        }
        friend float add(Box1,Box2);
};
class Box2{
    private:
    float len;
    float wid;
    float hei;
    public:
        BOX2()
        {
        len=3.0;
        wid=5.0;
        hei=4.0;
        }

        friend float add(Box1,Box2);
};
float add(Box1 b1,Box2 b2)
{
    float v1=b1.length*b1.height*b1.width;
    float v2=b2.len*b2.hei*b2.wid;
    return (v1+v2);
}
int main(int argc, char** argv) {
    Box1 b;
    Box2 b2;
    cout<<" Total volume is "<<add( b,b2);
}
```

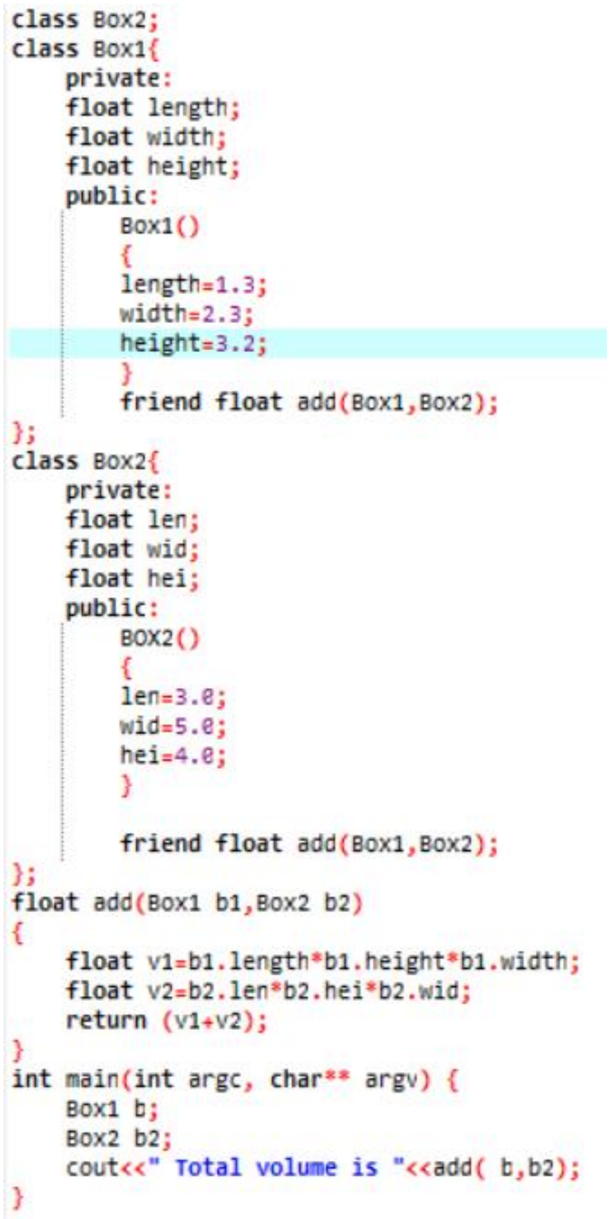


```
[Error] 'Box2' has not been declared
```

Credit: Khola Naseem

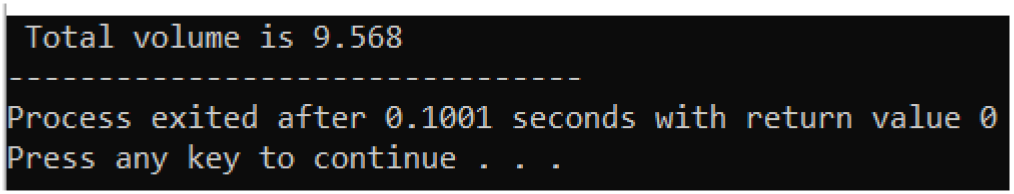
Friend function:

- Example:



```
class Box2;
class Box1{
    private:
    float length;
    float width;
    float height;
    public:
        Box1()
        {
        length=1.3;
        width=2.3;
        height=3.2;
        }
        friend float add(Box1,Box2);
};
class Box2{
    private:
    float len;
    float wid;
    float hei;
    public:
        BOX2()
        {
        len=3.0;
        wid=5.0;
        hei=4.0;
        }

        friend float add(Box1,Box2);
};
float add(Box1 b1,Box2 b2)
{
    float v1=b1.length*b1.height*b1.width;
    float v2=b2.len*b2.hei*b2.wid;
    return (v1+v2);
}
int main(int argc, char** argv) {
    Box1 b;
    Box2 b2;
    cout<<" Total volume is "<<add( b,b2);
}
```

Output:

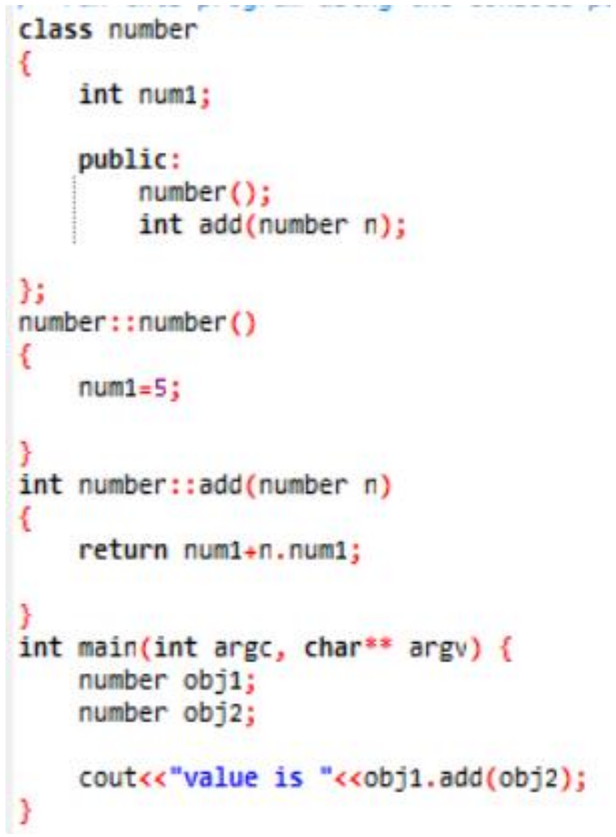


```
 Total volume is 9.568
--------------------------------
Process exited after 0.1001 seconds with return value 0
Press any key to continue . . .
```

Credit: Khola Naseem

Operator overloading :

- **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

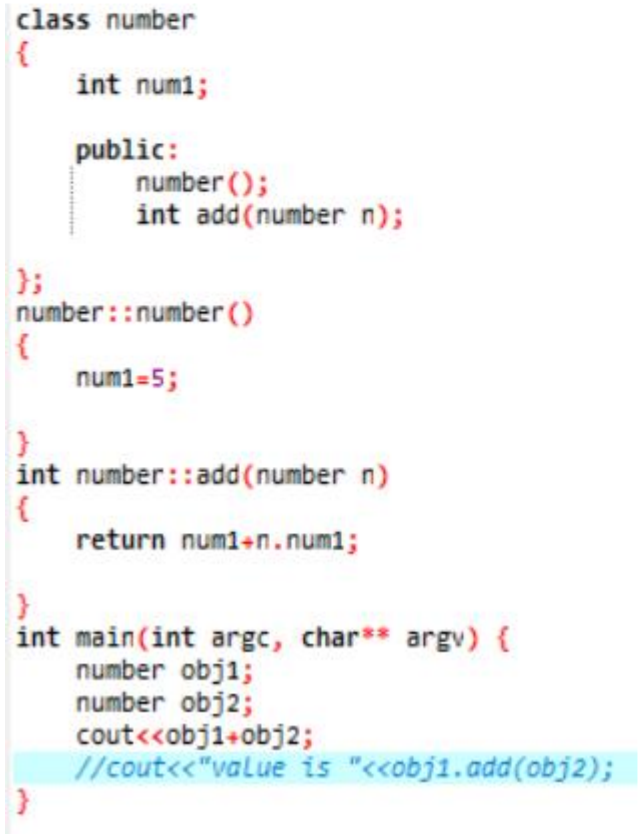


```
class number
{
    int num1;

    public:
        number();
        int add(number n);

};
number::number()
{
    num1=5;

}
int number::add(number n)
{
    return num1+n.num1;

}
int main(int argc, char** argv) {
    number obj1;
    number obj2;

    cout<<"value is "<<obj1.add(obj2);
}
```

Credit: Khola Naseem

Operator overloading:

- **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

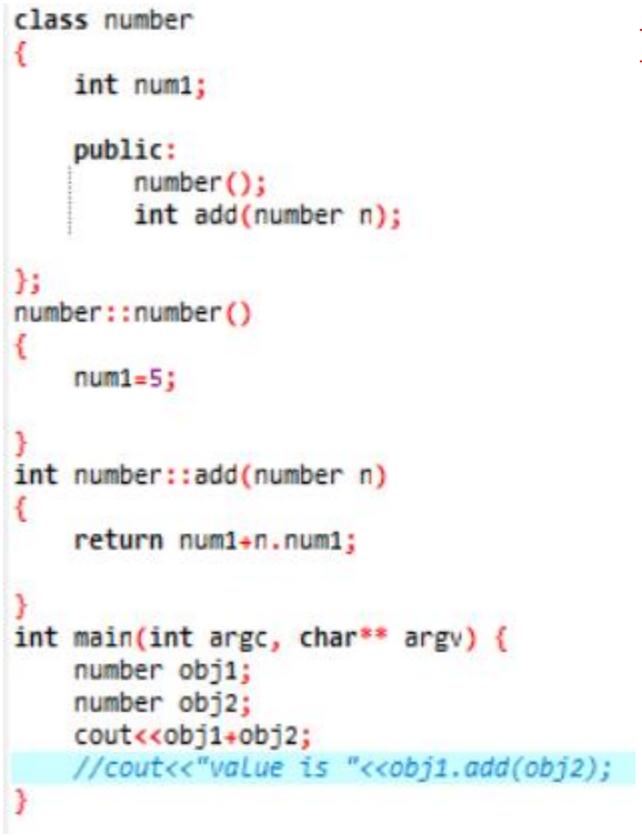


```
class number
{
    int num1;

    public:
        number();
        int add(number n);

};
number::number()
{
    num1=5;

}
int number::add(number n)
{
    return num1+n.num1;

}
int main(int argc, char** argv) {
    number obj1;
    number obj2;
    cout<<obj1+obj2;
    //cout<<"value is "<<obj1.add(obj2);
}
```

Credit: Khola Naseem

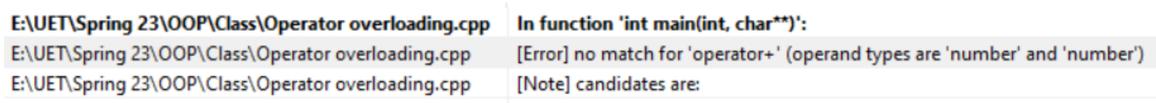
Operator overloading:

- **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
- **Why?**
-



```
class number
{
    int num1;

    public:
        number();
        int add(number n);

};
number::number()
{
    num1=5;

}
int number::add(number n)
{
    return num1+n.num1;

}
int main(int argc, char** argv) {
    number obj1;
    number obj2;
    cout<<obj1+obj2;
    //cout<<"value is "<<obj1.add(obj2);
}
```

Error

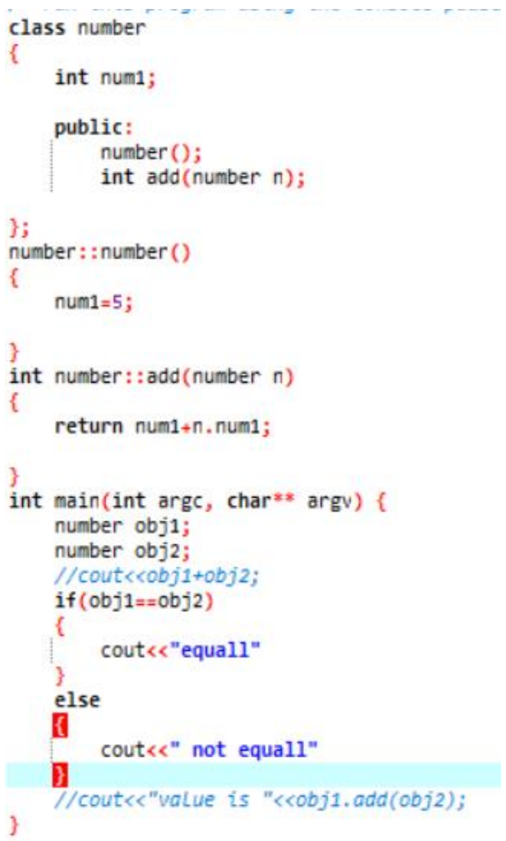


```
E:\UET\Spring 23\OOP\Class\Operator overloading.cpp   In function 'int main(int, char**)':
E:\UET\Spring 23\OOP\Class\Operator overloading.cpp   [Error] no match for 'operator+' (operand types are 'number' and 'number')
E:\UET\Spring 23\OOP\Class\Operator overloading.cpp   [Note] candidates are:
```

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

➢

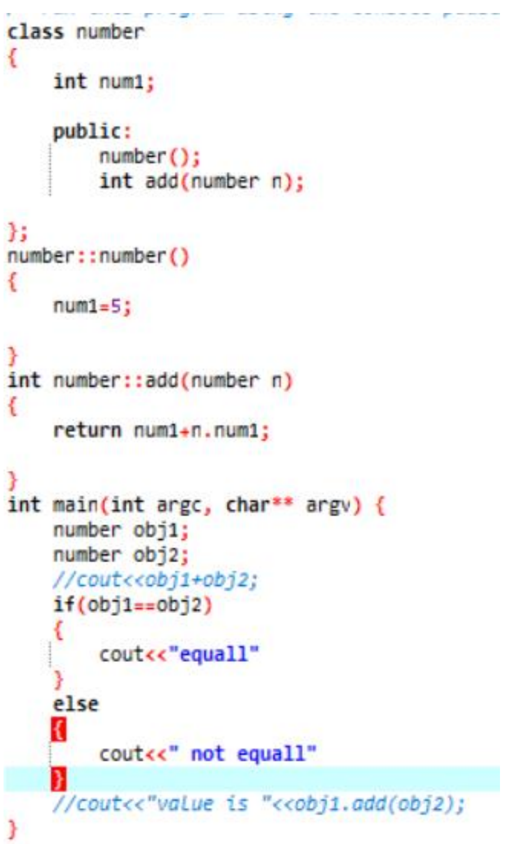


```
class number
{
    int num1;

    public:
        number();
        int add(number n);

};
number::number()
{
    num1=5;

}
int number::add(number n)
{
    return num1+n.num1;

}
int main(int argc, char** argv) {
    number obj1;
    number obj2;
    //cout<<obj1+obj2;
    if(obj1==obj2)
    {
        cout<<"equall"
    }
    else
    {
        cout<<" not equall"
    }
    //cout<<"value is "<<obj1.add(obj2);
}
```

Operator overloading:

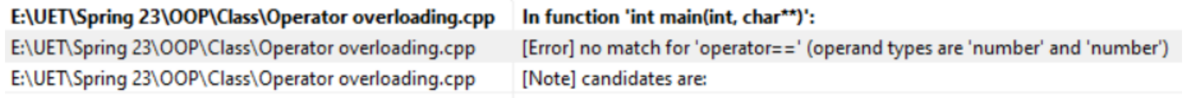
In C++, we can change the way operators work for user-defined types like objects and structures. This is known as operator overloading

➢



```
class number
{
    int num1;

    public:
        number();
        int add(number n);

};
number::number()
{
    num1=5;

}
int number::add(number n)
{
    return num1+n.num1;

}
int main(int argc, char** argv) {
    number obj1;
    number obj2;
    //cout<<obj1+obj2;
    if(obj1==obj2)
    {
        cout<<"equall"
    }
    else
    {
        cout<<" not equall"
    }
    //cout<<"value is "<<obj1.add(obj2);
}
```

Error

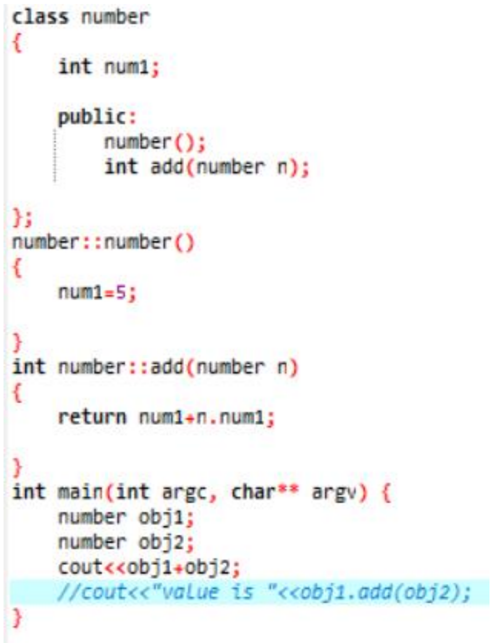


```
E:\UET\Spring 23\OOP\Class\Operator overloading.cpp    In function 'int main(int, char**)':
E:\UET\Spring 23\OOP\Class\Operator overloading.cpp    [Error] no match for 'operator==' (operand types are 'number' and 'number')
E:\UET\Spring 23\OOP\Class\Operator overloading.cpp    [Note] candidates are:
```

Credit: Khola Naseem

Operator overloading:

- **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
- Suppose we have created three objects obj1, obj2 and result from a class named number
- Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the number of obj1 and obj2 by writing the following code:



```
class number
{
    int num1;

    public:
        number();
        int add(number n);

};
number::number()
{
    num1=5;

}
int number::add(number n)
{
    return num1+n.num1;

}
int main(int argc, char** argv) {
    number obj1;
    number obj2;
    cout<<obj1+obj2;
    //cout<<"value is "<<obj1.add(obj2);
}
```

Credit: Khola Naseem

Operator overloading:

- **Operator overloading:**
- We cannot use operator overloading for fundamental data types like int, float, char and so on.
- C++ allows the programmer to extend the definitions of most of the operators so that operators such as relational operators, arithmetic operators, the insertion operator for data output, and the extraction operator for data input can be applied to classes. In C++ terminology, this is called operator overloading
- Some of the examples operator / works.(float,integer),<<,>>,+,-,*,<,>,==
- The only built-in operations on classes are assignment (=) and member selection. To use other operators on class objects, they must be explicitly overloaded.

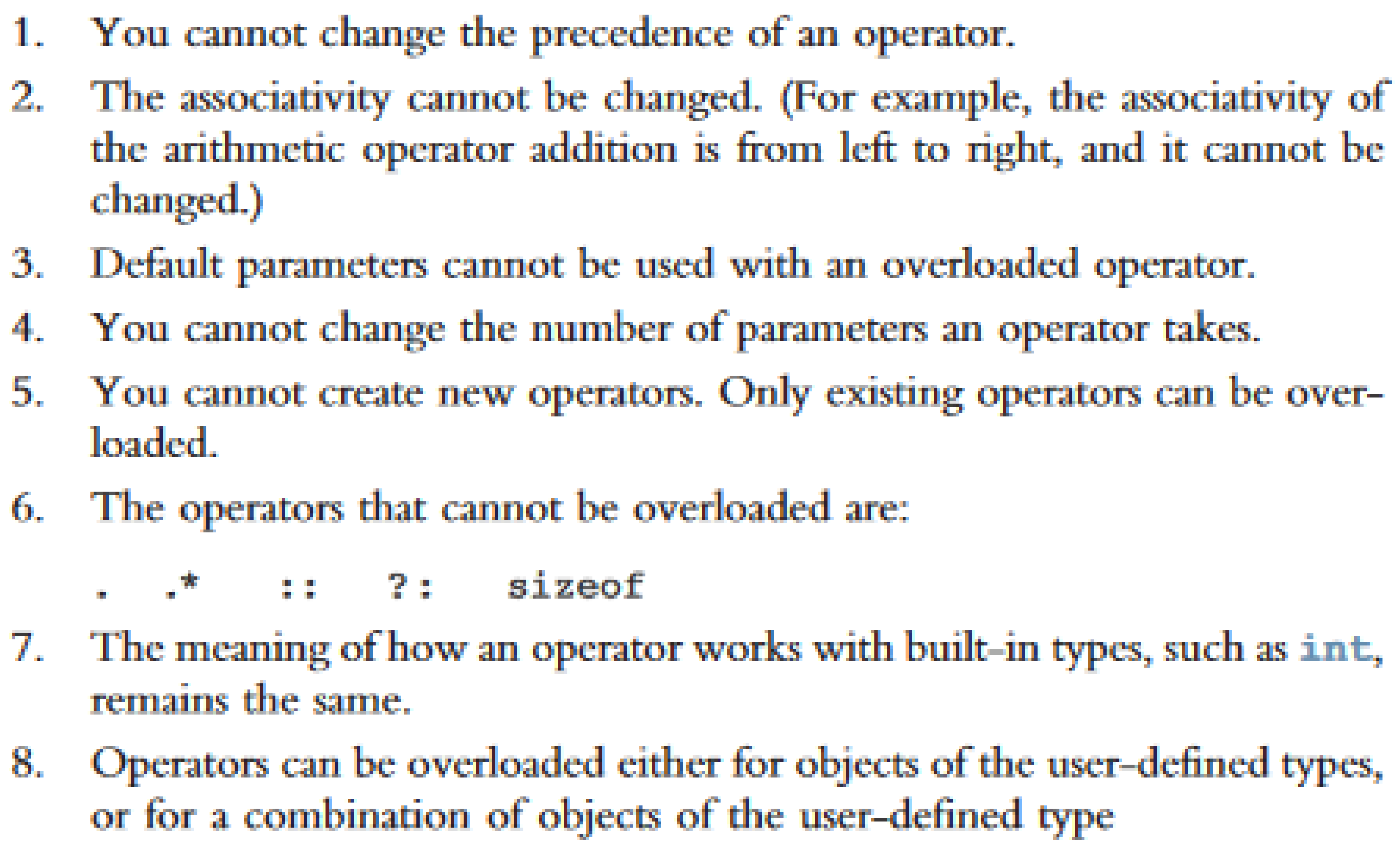
Operator overloading:

- **Operator overloading:**
- In order to overload operators, you must write functions (that is, the header and body).The name of the function that overloads an operator is the reserved word operator followed by the operator to be overloaded. For example
  - operator >=
- Syntax(header) for Operator Functions:

```
returnType operator operatorSymbol(formal parameter list)
```

Operator overloading:

➢ Overloading an Operator: Some Restrictions



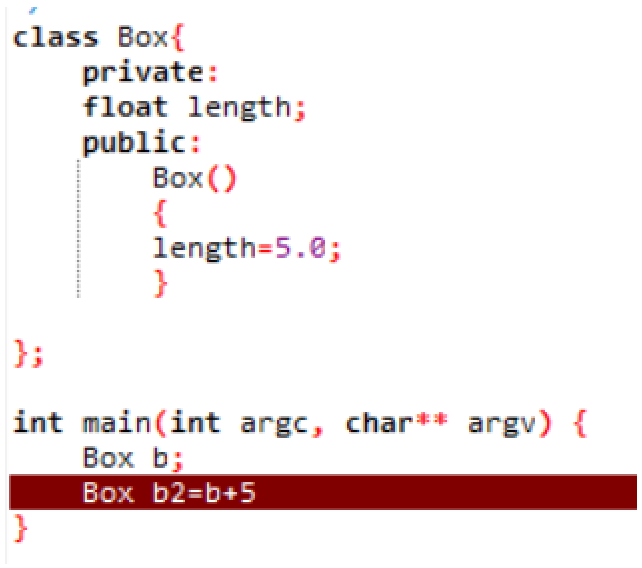
1. You cannot change the precedence of an operator.
2. The associativity cannot be changed. (For example, the associativity of the arithmetic operator addition is from left to right, and it cannot be changed.)
3. Default parameters cannot be used with an overloaded operator.
4. You cannot change the number of parameters an operator takes.
5. You cannot create new operators. Only existing operators can be overloaded.
6. The operators that cannot be overloaded are:

```
.   .*   ::   ?:   sizeof
```

7. The meaning of how an operator works with built-in types, such as `int`, remains the same.
8. Operators can be overloaded either for objects of the user-defined types, or for a combination of objects of the user-defined type

Operator overloading:

- Overloading an Operator: Some Restrictions

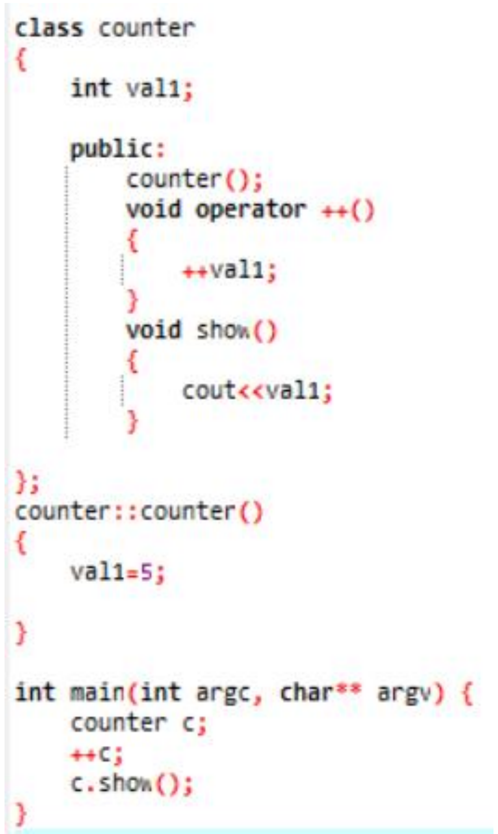


```
class Box{
    private:
    float length;
    public:
        Box()
        {
        length=5.0;
        }

};

int main(int argc, char** argv) {
    Box b;
    Box b2=b+5
}
```

Credit: Khola Naseem

Operator overloading:

- Operator Overloading in Unary Operators
- only one operand.
- Example:
  - The increment operator ++ and decrement operator –
  - ++ prefix:

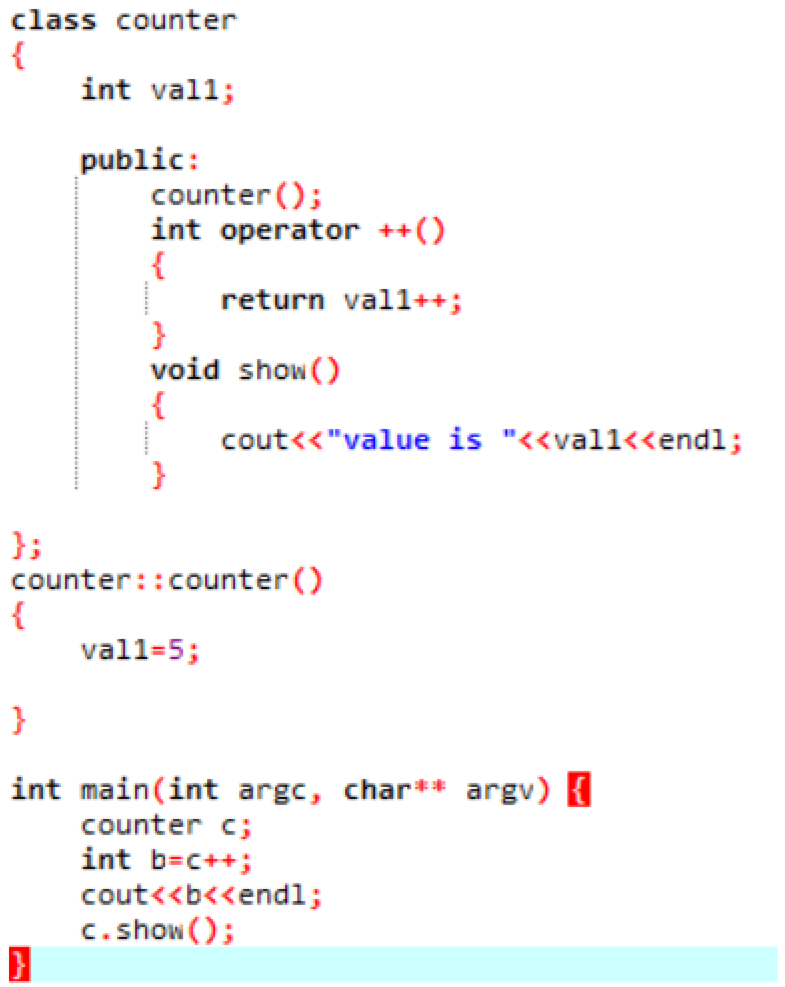


```
class counter
{
    int val1;

    public:
        counter();
        void operator ++()
        {
            ++val1;
        }
        void show()
        {
            cout<<val1;
        }

};
counter::counter()
{
    val1=5;

}

int main(int argc, char** argv) {
    counter c;
    ++c;
    c.show();
}
```

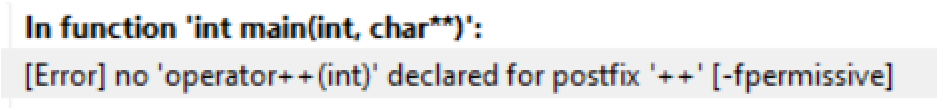
Credit: Khola Naseem

Operator overloading:

- Postfix++:



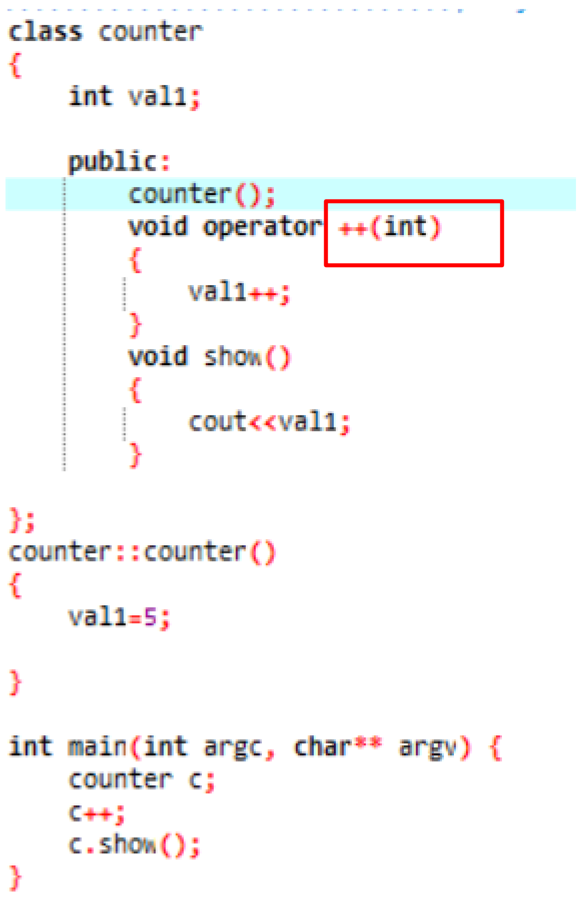
```
class counter
{
    int val1;

    public:
        counter();
        int operator ++()
        {
            return val1++;
        }
        void show()
        {
            cout<<"value is "<<val1<<endl;
        }

};
counter::counter()
{
    val1=5;

}

int main(int argc, char** argv) {
    counter c;
    int b=c++;
    cout<<b<<endl;
    c.show();
}
```



```
In function 'int main(int, char**)':
[Error] no 'operator++(int)' declared for postfix '++' [-fpermissive]
```

Operator overloading:

- Postfix++:



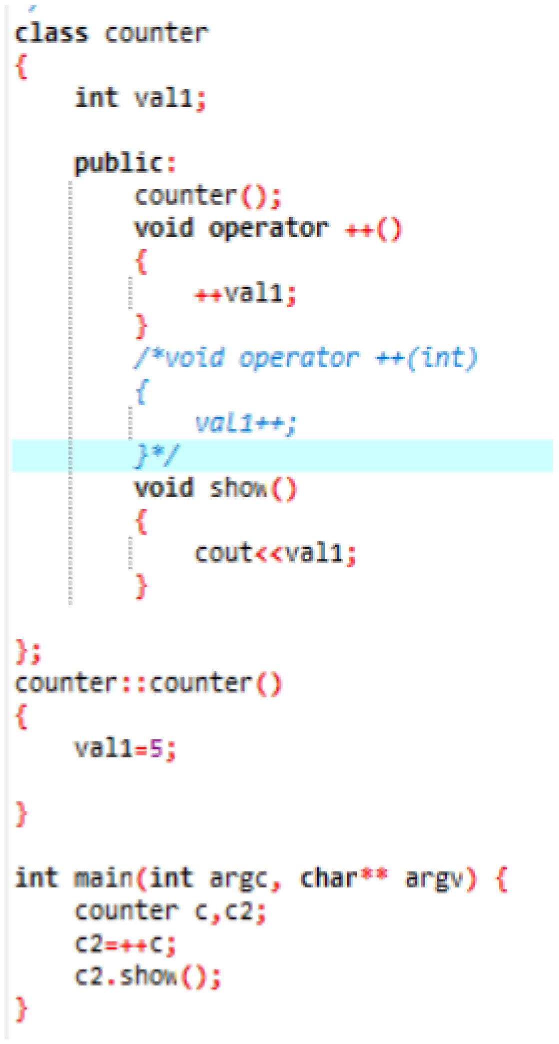
```
class counter
{
    int val1;

    public:
        counter();
        void operator ++(int)
        {
            val1++;
        }
        void show()
        {
            cout<<val1;
        }

};
counter::counter()
{
    val1=5;

}

int main(int argc, char** argv) {
    counter c;
    c++;
    c.show();
}
```

Credit: Khola Naseem

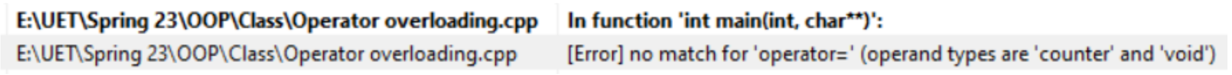
Operator overloading:

- unary:

Error:



```cpp
class counter
{
    int val1;

    public:
        counter();
        void operator ++()
        {
            ++val1;
        }
        /*void operator ++(int)
        {
            val1++;
        }*/
        void show()
        {
            cout<<val1;
        }

};
counter::counter()
{
    val1=5;

}

int main(int argc, char** argv) {
    counter c,c2;
    c2=++c;
    c2.show();
}
```

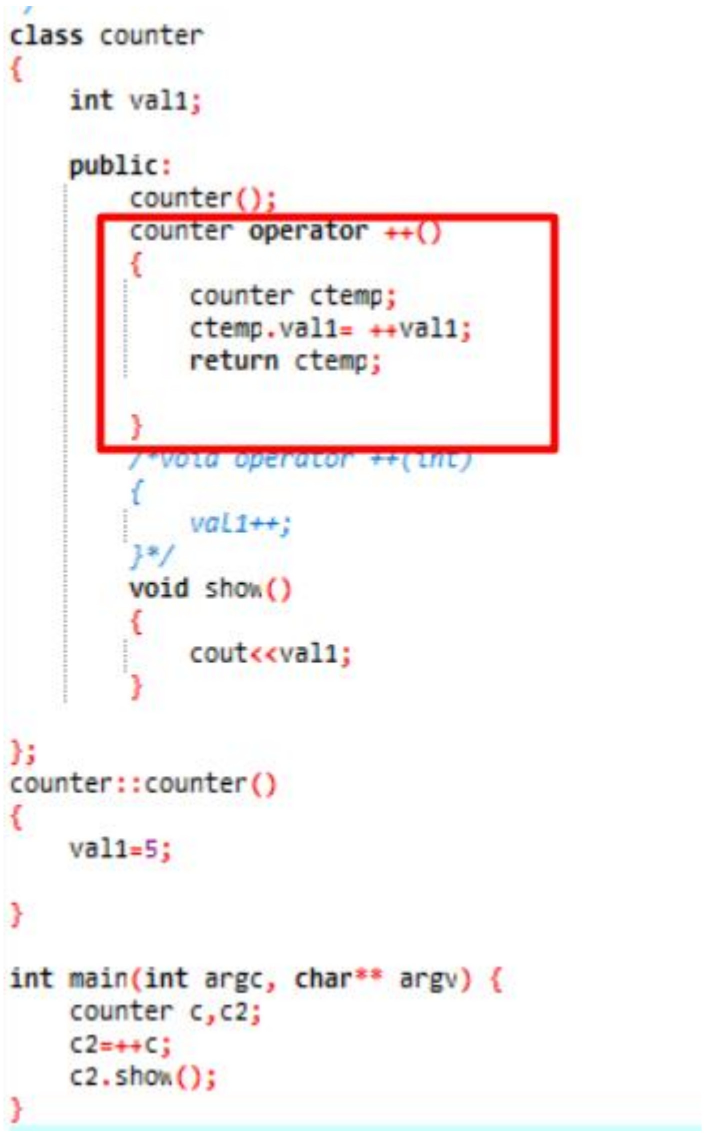


```
E:\UET\Spring 23\OOP\Class\Operator overloading.cpp    In function 'int main(int, char**)':
E:\UET\Spring 23\OOP\Class\Operator overloading.cpp    [Error] no match for 'operator=' (operand types are 'counter' and 'void')
```

Credit: Khola Naseem

Operator overloading:

- unary:

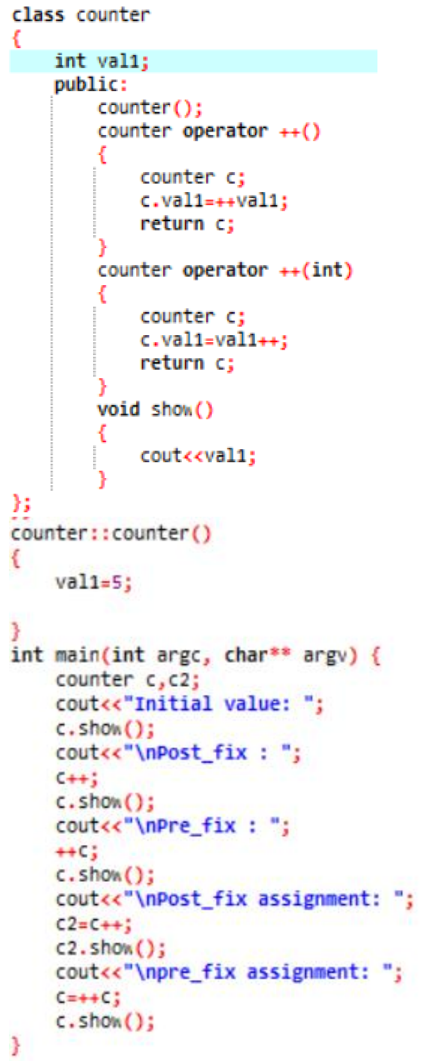


```
class counter
{
    int val1;

    public:
        counter();
        counter operator ++()
        {
            counter ctemp;
            ctemp.val1= ++val1;
            return ctemp;

        }
        /*void operator ++(int)
        {
            val1++;
        }*/
        void show()
        {
            cout<<val1;
        }

};
counter::counter()
{
    val1=5;

}

int main(int argc, char** argv) {
    counter c,c2;
    c2=++c;
    c2.show();
}
```

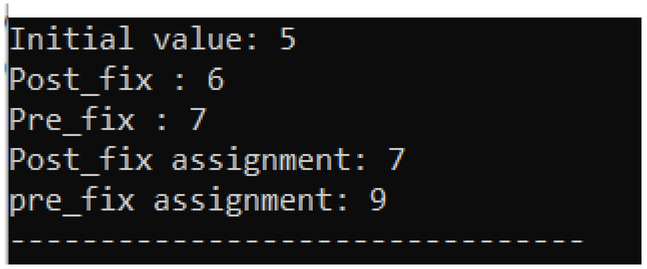
Credit: Khola Naseem

Operator overloading:

- unary:



```
class counter
{
    int val1;
    public:
        counter();
        counter operator ++()
        {
            counter c;
            c.val1=++val1;
            return c;
        }
        counter operator ++(int)
        {
            counter c;
            c.val1=val1++;
            return c;
        }
        void show()
        {
            cout<<val1;
        }
};
counter::counter()
{
    val1=5;

}
int main(int argc, char** argv) {
    counter c,c2;
    cout<<"Initial value: ";
    c.show();
    cout<<"\nPost_fix : ";
    c++;
    c.show();
    cout<<"\nPre_fix : ";
    ++c;
    c.show();
    cout<<"\nPost_fix assignment: ";
    c2=c++;
    c2.show();
    cout<<"\npre_fix assignment: ";
    c=++c;
    c.show();
}
```



```
Initial value: 5
Post_fix : 6
Pre_fix : 7
Post_fix assignment: 7
pre_fix assignment: 9
---------------------------------
```

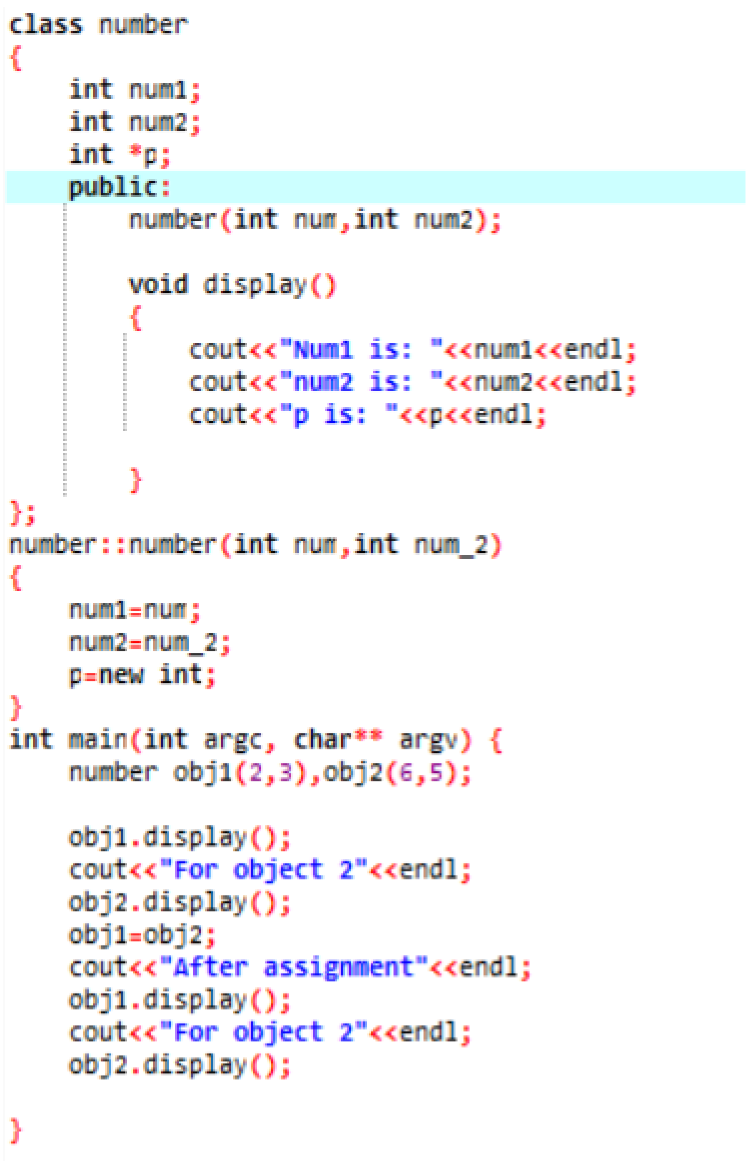
Credit: Khola Naseem

Operator overloading:

- Task:
  - Implement the pre and post decrement operators

Operator overloading:

- unary: **Assignment**

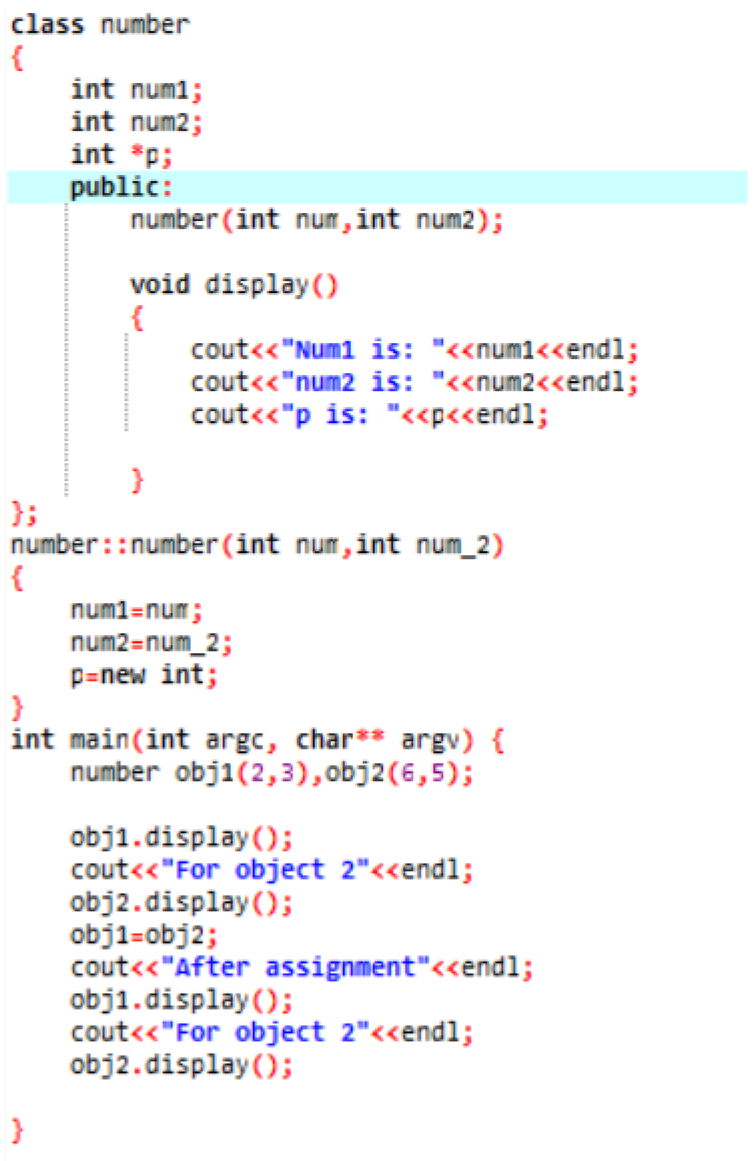


```
class number
{
    int num1;
    int num2;
    int *p;
    public:
        number(int num,int num2);

        void display()
        {
            cout<<"Num1 is: "<<num1<<endl;
            cout<<"num2 is: "<<num2<<endl;
            cout<<"p is: "<<p<<endl;

        }
};
number::number(int num,int num_2)
{
    num1=num;
    num2=num_2;
    p=new int;
}
int main(int argc, char** argv) {
    number obj1(2,3),obj2(6,5);

    obj1.display();
    cout<<"For object 2"<<endl;
    obj2.display();
    obj1=obj2;
    cout<<"After assignment"<<endl;
    obj1.display();
    cout<<"For object 2"<<endl;
    obj2.display();

}
```

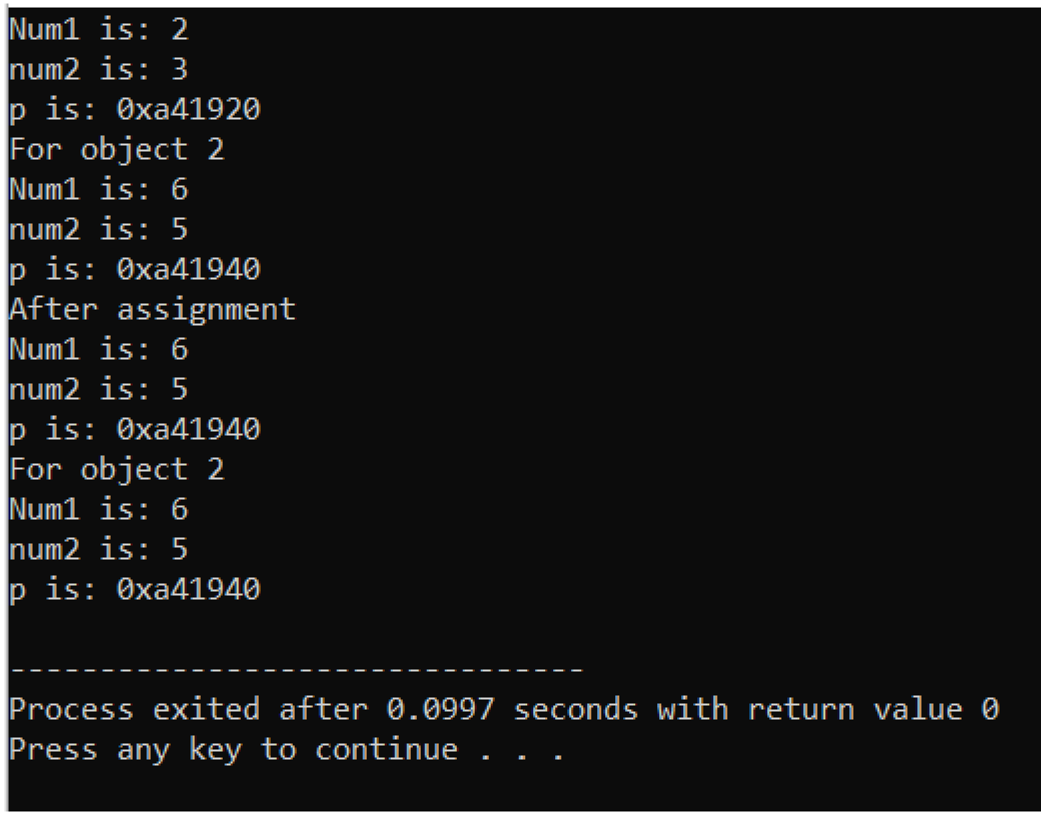
Credit: Khola Naseem

Operator overloading:

- unary: **Assignment** **output**



```
class number
{
    int num1;
    int num2;
    int *p;
    public:
        number(int num,int num2);

        void display()
        {
            cout<<"Num1 is: "<<num1<<endl;
            cout<<"num2 is: "<<num2<<endl;
            cout<<"p is: "<<p<<endl;

        }
};
number::number(int num,int num_2)
{
    num1=num;
    num2=num_2;
    p=new int;
}
int main(int argc, char** argv) {
    number obj1(2,3),obj2(6,5);

    obj1.display();
    cout<<"For object 2"<<endl;
    obj2.display();
    obj1=obj2;
    cout<<"After assignment"<<endl;
    obj1.display();
    cout<<"For object 2"<<endl;
    obj2.display();

}
```

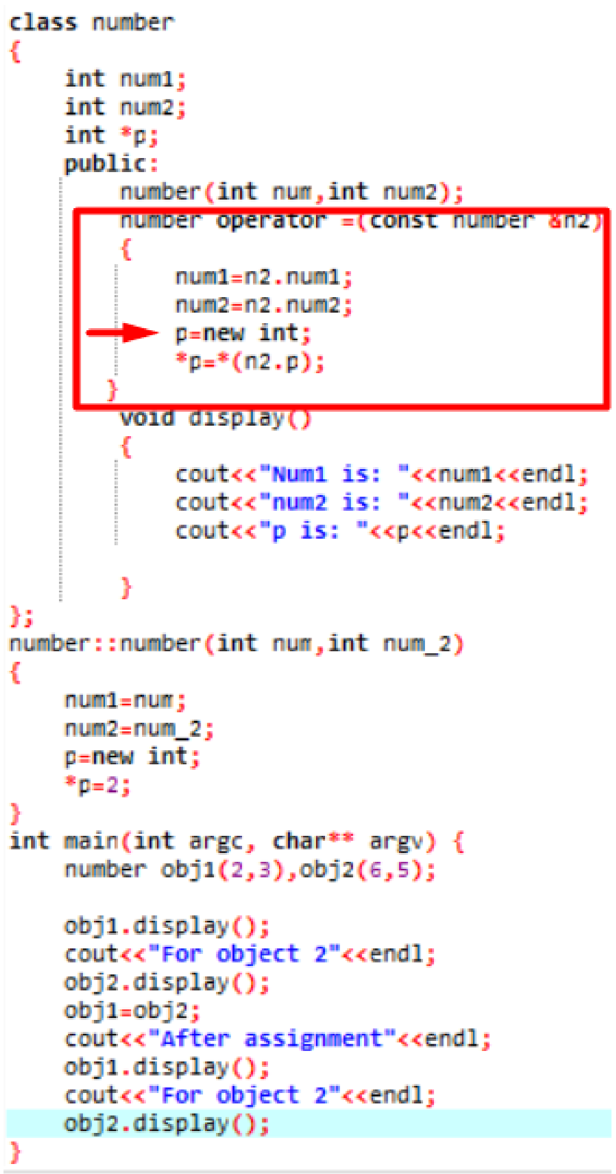


```
Num1 is: 2
num2 is: 3
p is: 0xa41920
For object 2
Num1 is: 6
num2 is: 5
p is: 0xa41940
After assignment
Num1 is: 6
num2 is: 5
p is: 0xa41940
For object 2
Num1 is: 6
num2 is: 5
p is: 0xa41940


--------------------------------
Process exited after 0.0997 seconds with return value 0
Press any key to continue . . .
```

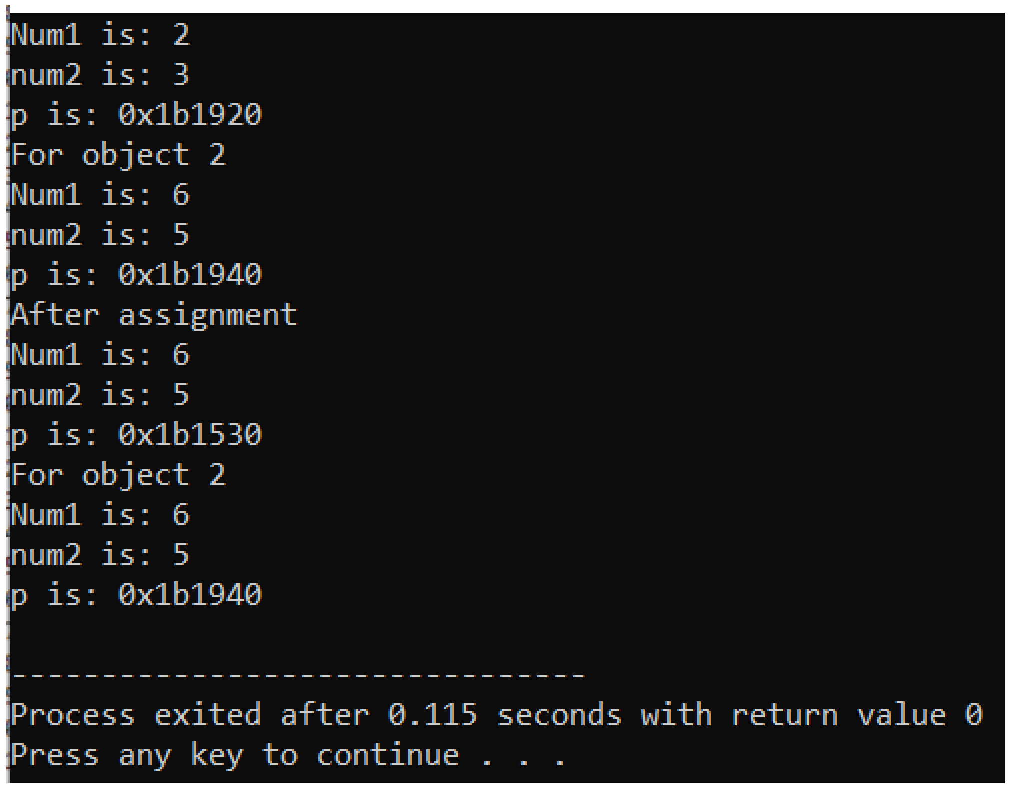
Credit: Khola Naseem

Operator overloading:

- unary: **Assignment** **output**



```
class number
{
    int num1;
    int num2;
    int *p;
    public:
        number(int num,int num2);
        number operator =(const number &n2)
        {
            num1=n2.num1;
            num2=n2.num2;
            p=new int;
            *p=*(n2.p);
        }
        void display()
        {
            cout<<"Num1 is: "<<num1<<endl;
            cout<<"num2 is: "<<num2<<endl;
            cout<<"p is: "<<p<<endl;

        }
};
number::number(int num,int num_2)
{
    num1=num;
    num2=num_2;
    p=new int;
    *p=2;
}
int main(int argc, char** argv) {
    number obj1(2,3),obj2(6,5);

    obj1.display();
    cout<<"For object 2"<<endl;
    obj2.display();
    obj1=obj2;
    cout<<"After assignment"<<endl;
    obj1.display();
    cout<<"For object 2"<<endl;
    obj2.display();
}
```

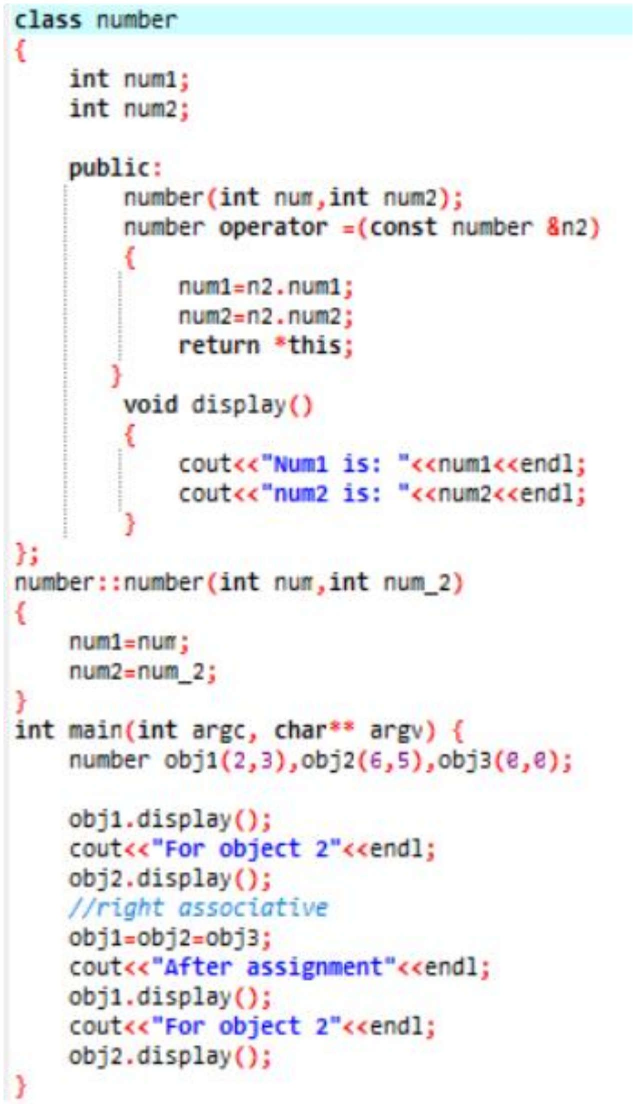


```
Num1 is: 2
num2 is: 3
p is: 0x1b1920
For object 2
Num1 is: 6
num2 is: 5
p is: 0x1b1940
After assignment
Num1 is: 6
num2 is: 5
p is: 0x1b1530
For object 2
Num1 is: 6
num2 is: 5
p is: 0x1b1940

--------------------------------
Process exited after 0.115 seconds with return value 0
Press any key to continue . . .
```

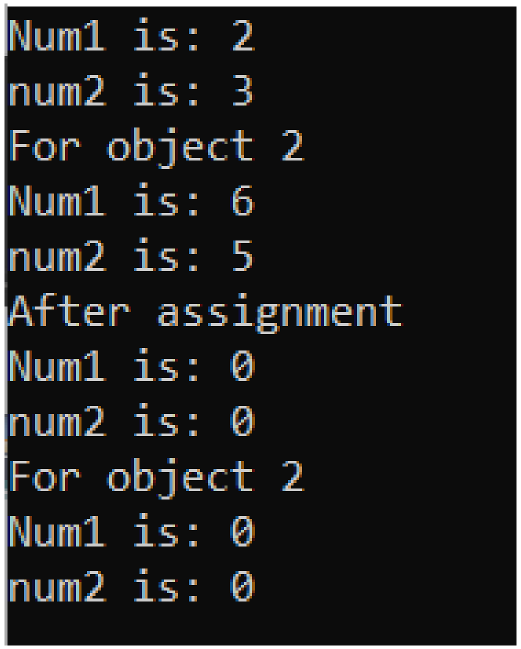
Credit: Khola Naseem

Operator overloading:

- Assignment Operator



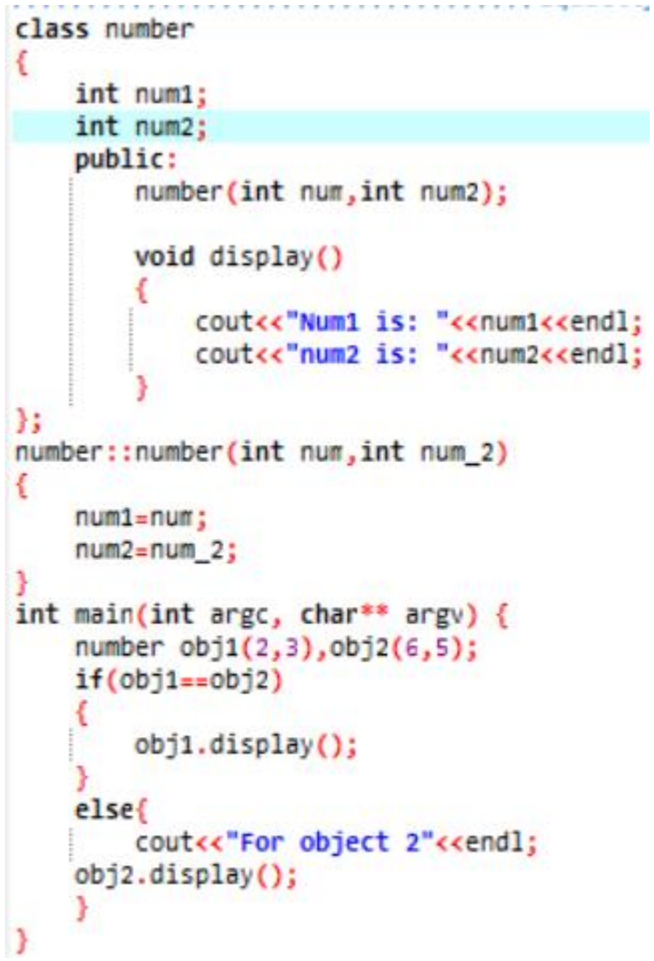
```
class number
{
    int num1;
    int num2;

    public:
        number(int num,int num2);
        number operator =(const number &n2)
        {
            num1=n2.num1;
            num2=n2.num2;
            return *this;
        }
        void display()
        {
            cout<<"Num1 is: "<<num1<<endl;
            cout<<"num2 is: "<<num2<<endl;
        }
};
number::number(int num,int num_2)
{
    num1=num;
    num2=num_2;
}
int main(int argc, char** argv) {
    number obj1(2,3),obj2(6,5),obj3(0,0);

    obj1.display();
    cout<<"For object 2"<<endl;
    obj2.display();
    //right associative
    obj1=obj2=obj3;
    cout<<"After assignment"<<endl;
    obj1.display();
    cout<<"For object 2"<<endl;
    obj2.display();
}
```



```
Num1 is: 2
num2 is: 3
For object 2
Num1 is: 6
num2 is: 5
After assignment
Num1 is: 0
num2 is: 0
For object 2
Num1 is: 0
num2 is: 0
```

Overloading Relational Operators:

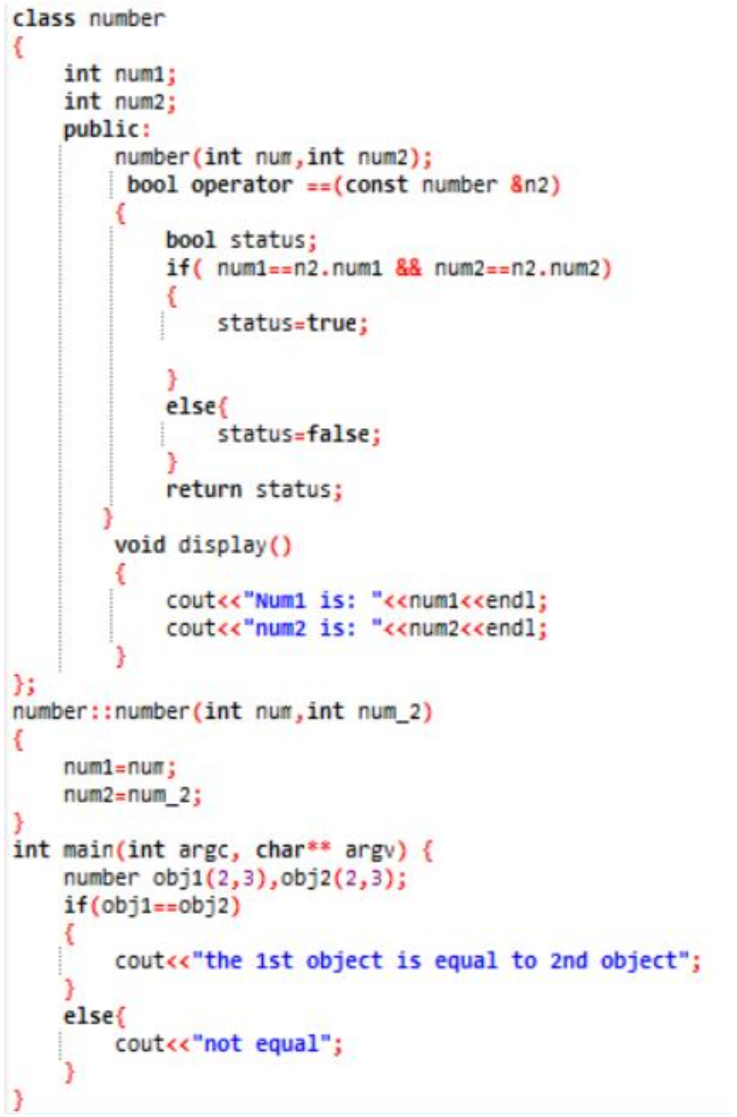
- ==
- Error



```
class number
{
    int num1;
    int num2;
    public:
        number(int num,int num2);

        void display()
        {
            cout<<"Num1 is: "<<num1<<endl;
            cout<<"num2 is: "<<num2<<endl;
        }
};
number::number(int num,int num_2)
{
    num1=num;
    num2=num_2;
}
int main(int argc, char** argv) {
    number obj1(2,3),obj2(6,5);
    if(obj1==obj2)
    {
        obj1.display();
    }
    else{
        cout<<"For object 2"<<endl;
    obj2.display();
    }
}
```

Operator overloading:

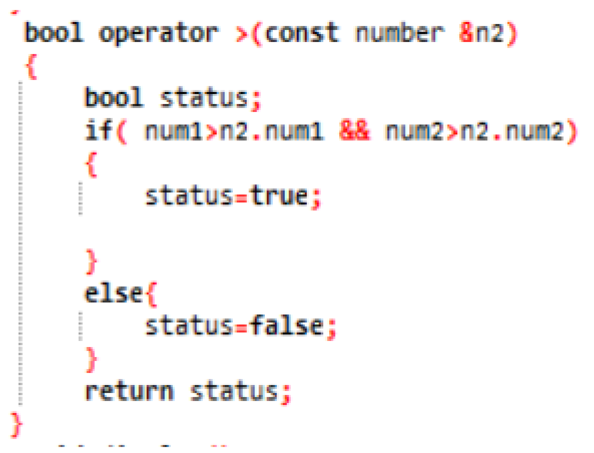
- ==



```
class number
{
    int num1;
    int num2;
    public:
        number(int num,int num2);
         bool operator ==(const number &n2)
        {
            bool status;
            if( num1==n2.num1 && num2==n2.num2)
            {
                status=true;

            }
            else{
                status=false;
            }
            return status;
        }
        void display()
        {
            cout<<"Num1 is: "<<num1<<endl;
            cout<<"num2 is: "<<num2<<endl;
        }
};
number::number(int num,int num_2)
{
    num1=num;
    num2=num_2;
}
int main(int argc, char** argv) {
    number obj1(2,3),obj2(2,3);
    if(obj1==obj2)
    {
        cout<<"the 1st object is equal to 2nd object";
    }
    else{
        cout<<"not equal";
    }
}
```

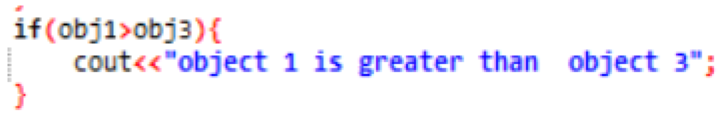
Overloading Relational Operators:

- >



```
bool operator >(const number &n2)
{
    bool status;
    if( num1>n2.num1 && num2>n2.num2)
    {
        status=true;

    }
    else{
        status=false;
    }
    return status;
}
```

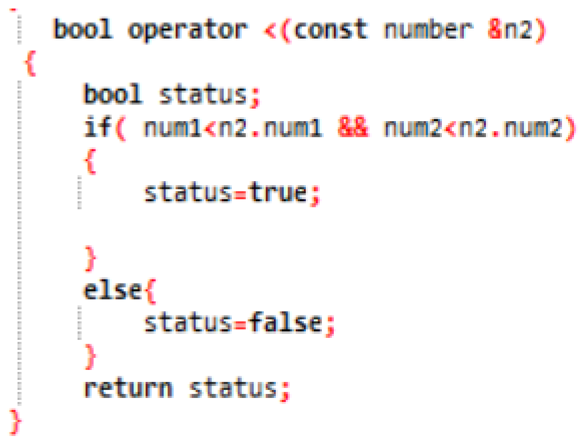
- Check



```
if(obj1>obj3){
    cout<<"object 1 is greater than  object 3";
}
```

Overloading Relational Operators:

- <

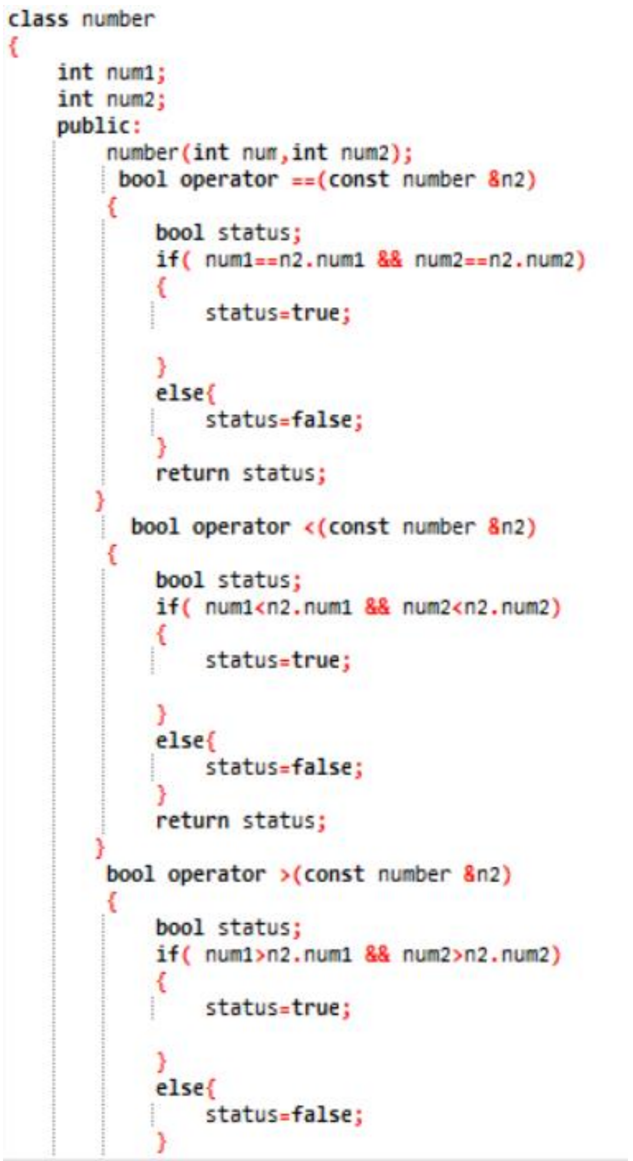


```
bool operator <(const number &n2)
{
    bool status;
    if( num1<n2.num1 && num2<n2.num2)
    {
        status=true;

    }
    else{
        status=false;
    }
    return status;
}
```

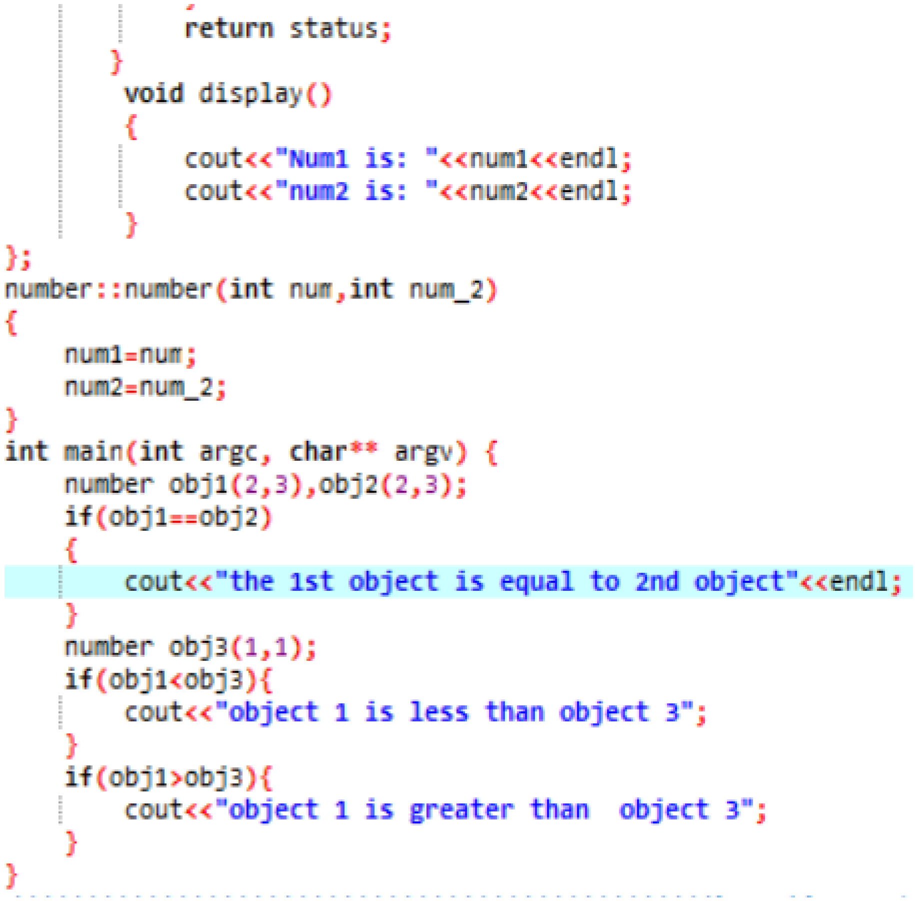
- Check

Overloading Relational Operators:

- <, >, =



```
class number
{
    int num1;
    int num2;
    public:
        number(int num,int num2);
         bool operator ==(const number &n2)
        {
            bool status;
            if( num1==n2.num1 && num2==n2.num2)
            {
                status=true;

            }
            else{
                status=false;
            }
            return status;
        }
         bool operator <(const number &n2)
        {
            bool status;
            if( num1<n2.num1 && num2<n2.num2)
            {
                status=true;

            }
            else{
                status=false;
            }
            return status;
        }
        bool operator >(const number &n2)
        {
            bool status;
            if( num1>n2.num1 && num2>n2.num2)
            {
                status=true;

            }
            else{
                status=false;
            }
```



```
            return status;
        }
        void display()
        {
            cout<<"Num1 is: "<<num1<<endl;
            cout<<"num2 is: "<<num2<<endl;
        }
};
number::number(int num,int num_2)
{
    num1=num;
    num2=num_2;
}
int main(int argc, char** argv) {
    number obj1(2,3),obj2(2,3);
    if(obj1==obj2)
    {
        cout<<"the 1st object is equal to 2nd object"<<endl;
    }
    number obj3(1,1);
    if(obj1<obj3){
        cout<<"object 1 is less than object 3";
    }
    if(obj1>obj3){
        cout<<"object 1 is greater than  object 3";
    }
}
```

Credit: Khola Naseem

Reference material

- **For Practice Questions, refer to these books**
    - C++ Programming From Problem Analysis To Program Design, 5th Edition, D.S.Malik. Chapter 12.
    - C++ How to Program, Deitel & Deitel, 5th Edition, Prentice Hall.
    - Object Oriented Programming in C++ by Robert Lafore.
    - Object Oriented Software Construction, Bertrand Meyer's
    - Object-Oriented Analysis and Design with applications, Grady Booch et al, 3Rd Edition, Pearson, 2007
    - Web