Operator Overloading

CSE 205 - Week 11 Class 2 & 3

Lec Raiyan Rahman
Dept of CSE, MIST
raiyan@cse.mist.ac.bd
CC: Lec Saidul Hoque Anik

Remember Function Overloading?

Same function name, different definition **based on parameter**.

```
void printMax(int a, int b);

void printMax(double a, double b);
```

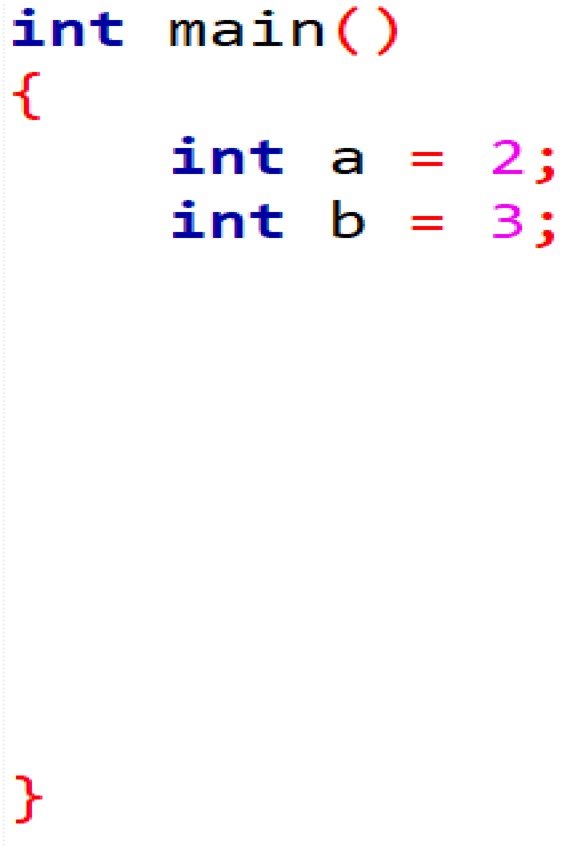
But Can We Do This?

```
int a = 10;
int b = 20;

int res;

res = a + b;

cout<<res; //30
```

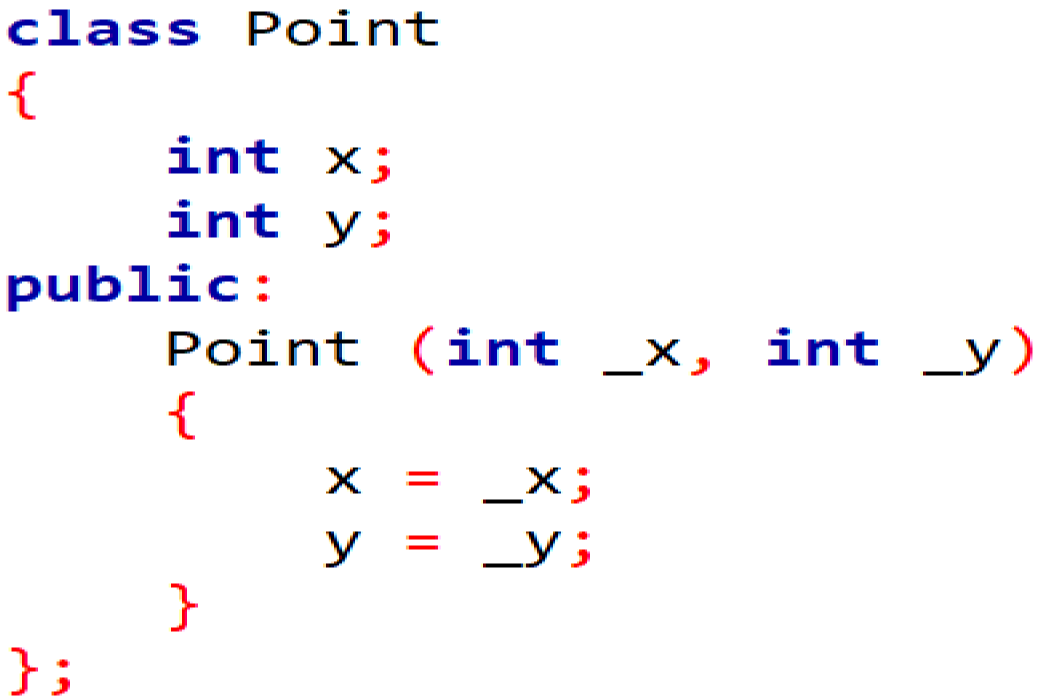
```
Point P1(1, 1);
Point P2 (2, 2);

Point res;

res = P1 + P2;

cout<<res; //(3, 3)
```

Solution: Operator Overloading

Same operator, different definition **based on parameter**.



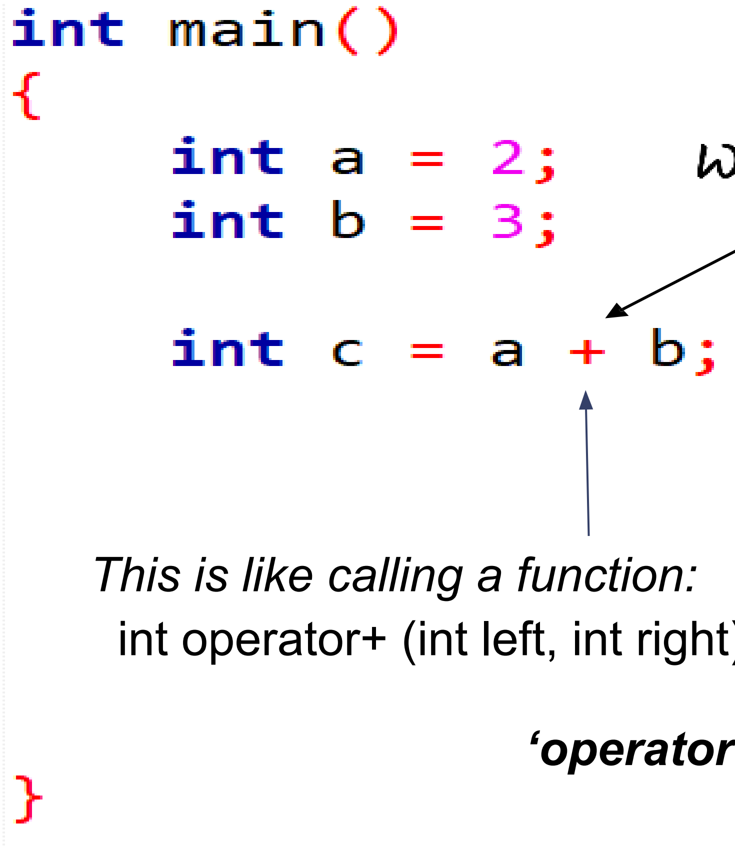
```
int main()
{
    int a = 2;
    int b = 3;




}
```



```
class Point
{
    int x;
    int y;
public:
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
};
```

Operator Overloading

Same operator, different definition **based on parameter**.



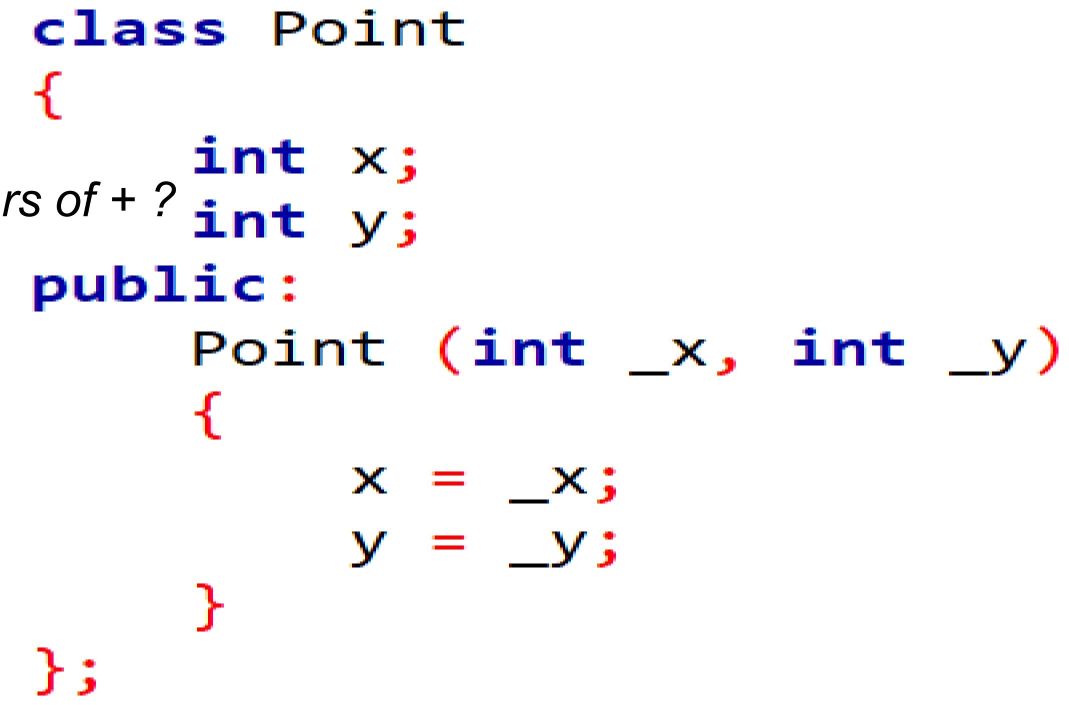
```
int main()
{
	int a = 2;
	int b = 3;

	int c = a + b;



}
```

What are the parameters of + ?

This is like calling a function:

int operator+ (int left, int right)

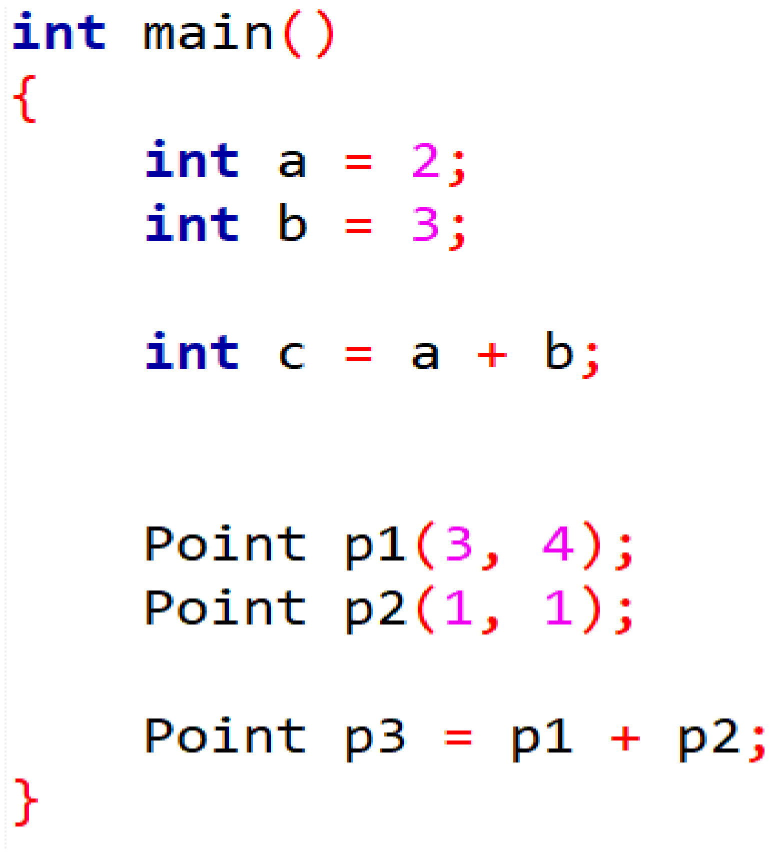
'operator' is a keyword



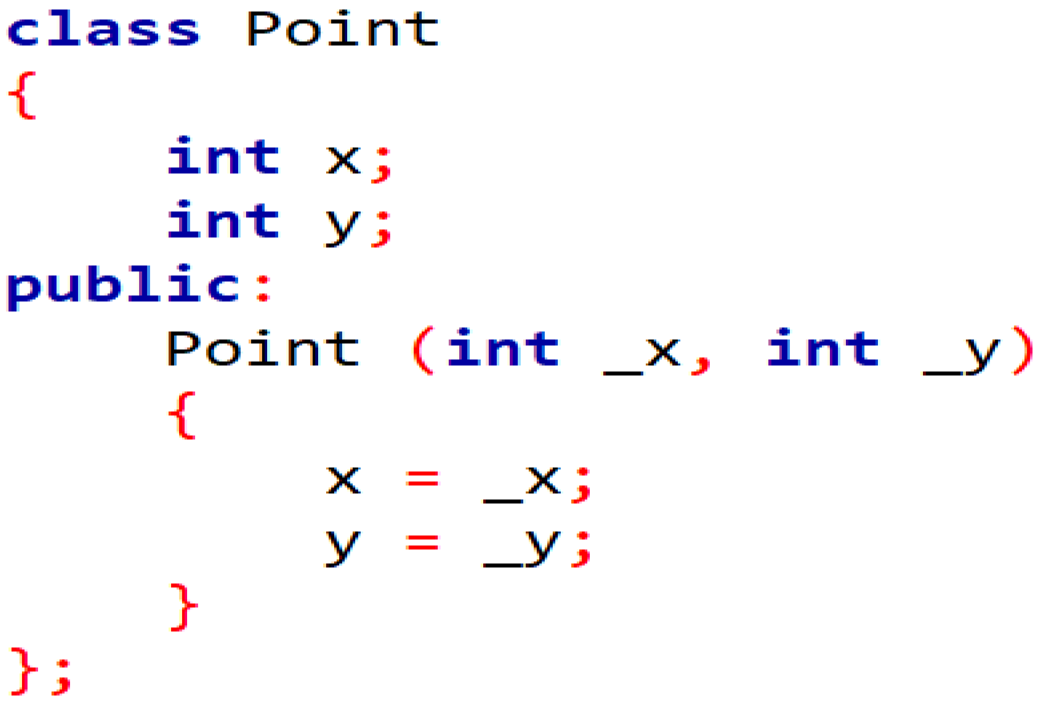
```
class Point
{
	int x;
	int y;
public:
	Point (int _x, int _y)
	{
		x = _x;
		y = _y;
	}
};
```

Operator Overloading

Same operator, different definition **based on parameter**.



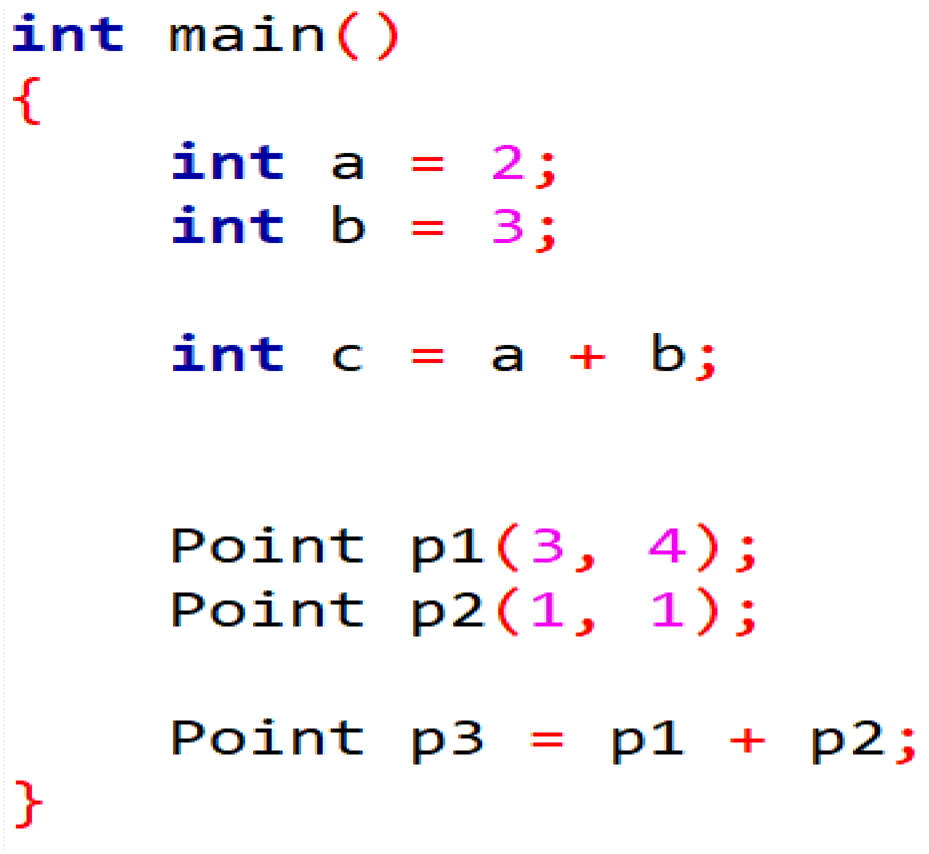
```
int main()
{
    int a = 2;
    int b = 3;

    int c = a + b;


    Point p1(3, 4);
    Point p2(1, 1);

    Point p3 = p1 + p2;
}
```



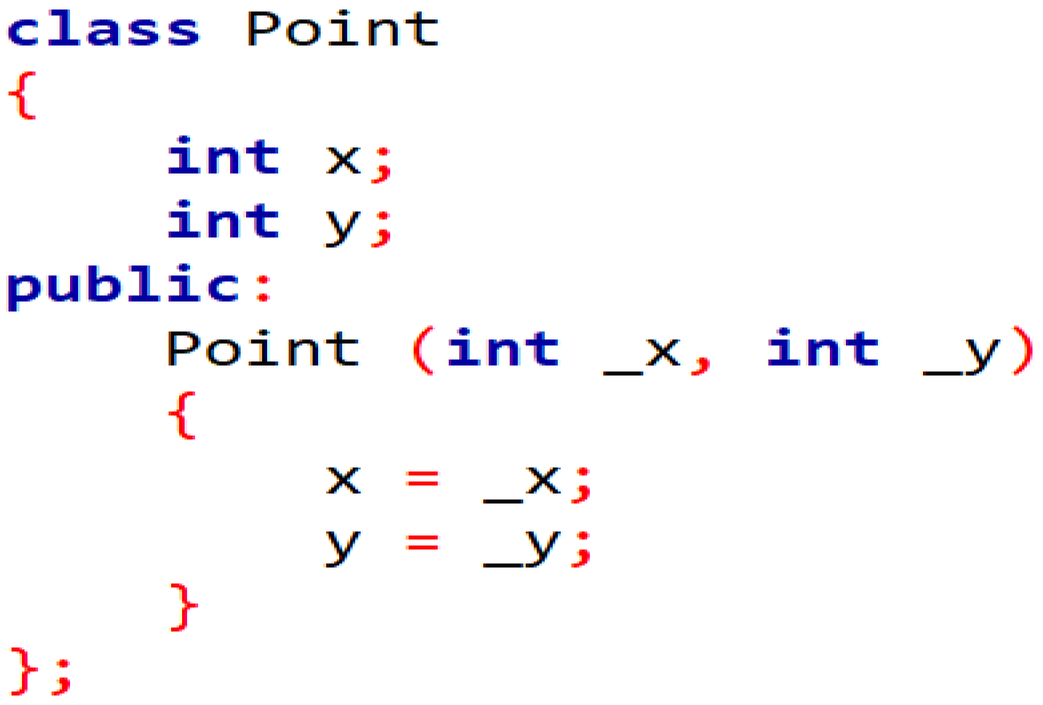
```
class Point
{
    int x;
    int y;
public:
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
};
```

Operator Overloading

Same operator, different definition **based on parameter**.



```
int main()
{
    int a = 2;
    int b = 3;

    int c = a + b;


    Point p1(3, 4);
    Point p2(1, 1);

    Point p3 = p1 + p2;
}
```

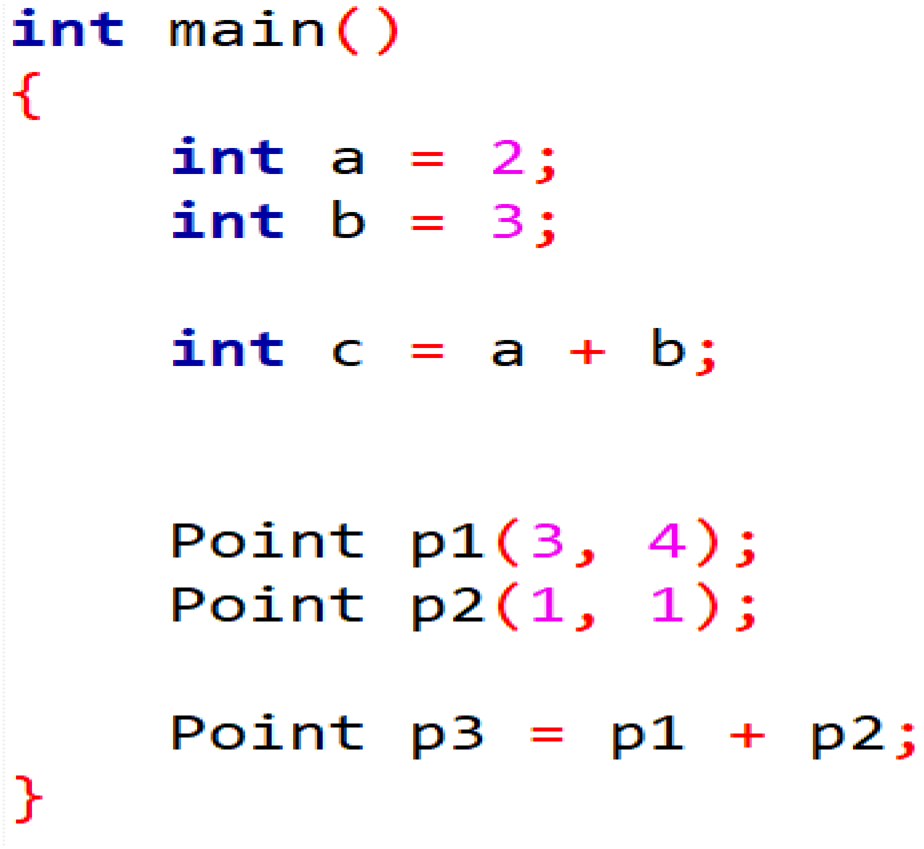


```
class Point
{
    int x;
    int y;
public:
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
};
```

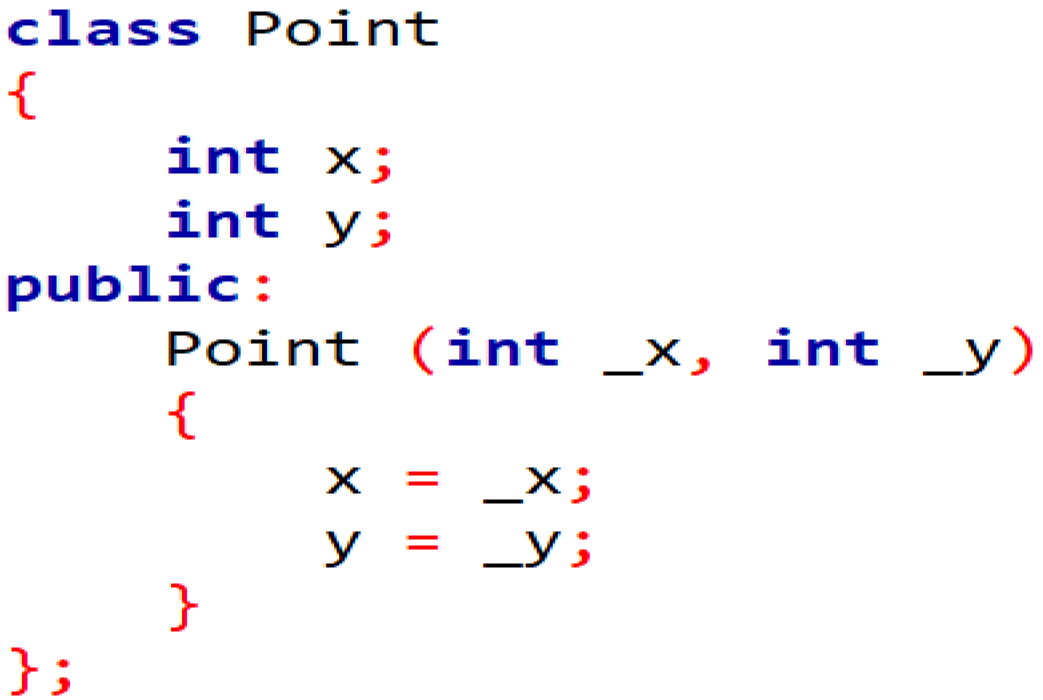
//p3 = (4, 5)

Operator Overloading

Same operator, different definition **based on parameter**.



```
int main()
{
    int a = 2;
    int b = 3;

    int c = a + b;


    Point p1(3, 4);
    Point p2(1, 1);

    Point p3 = p1 + p2;
}
```

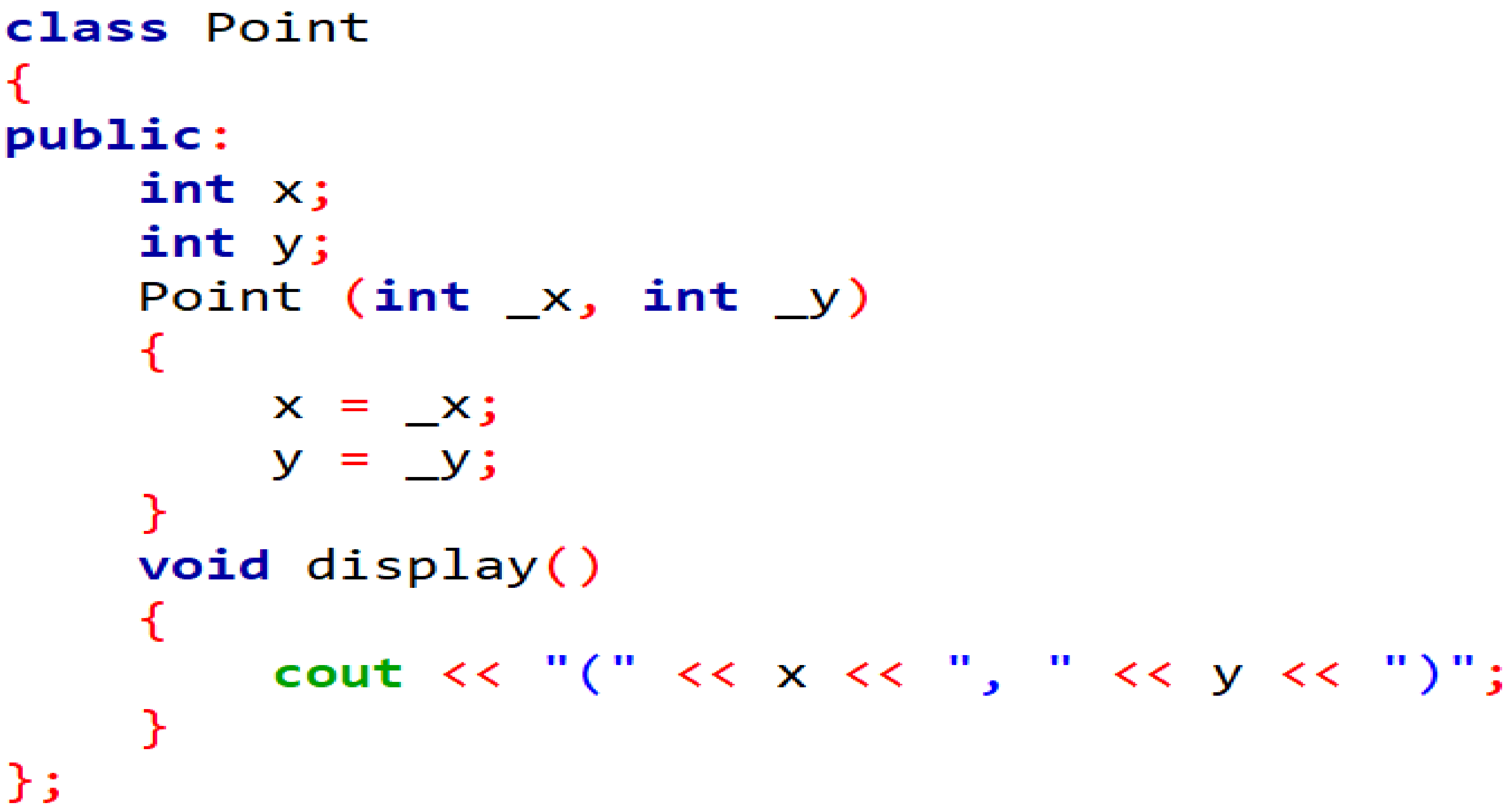


```
class Point
{
    int x;
    int y;
public:
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
};
```

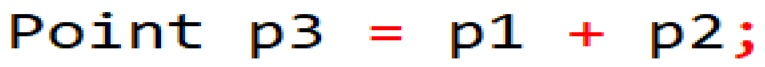
what will be the overloaded function?

operator+ (Point left, Point right)

Operator Overloading

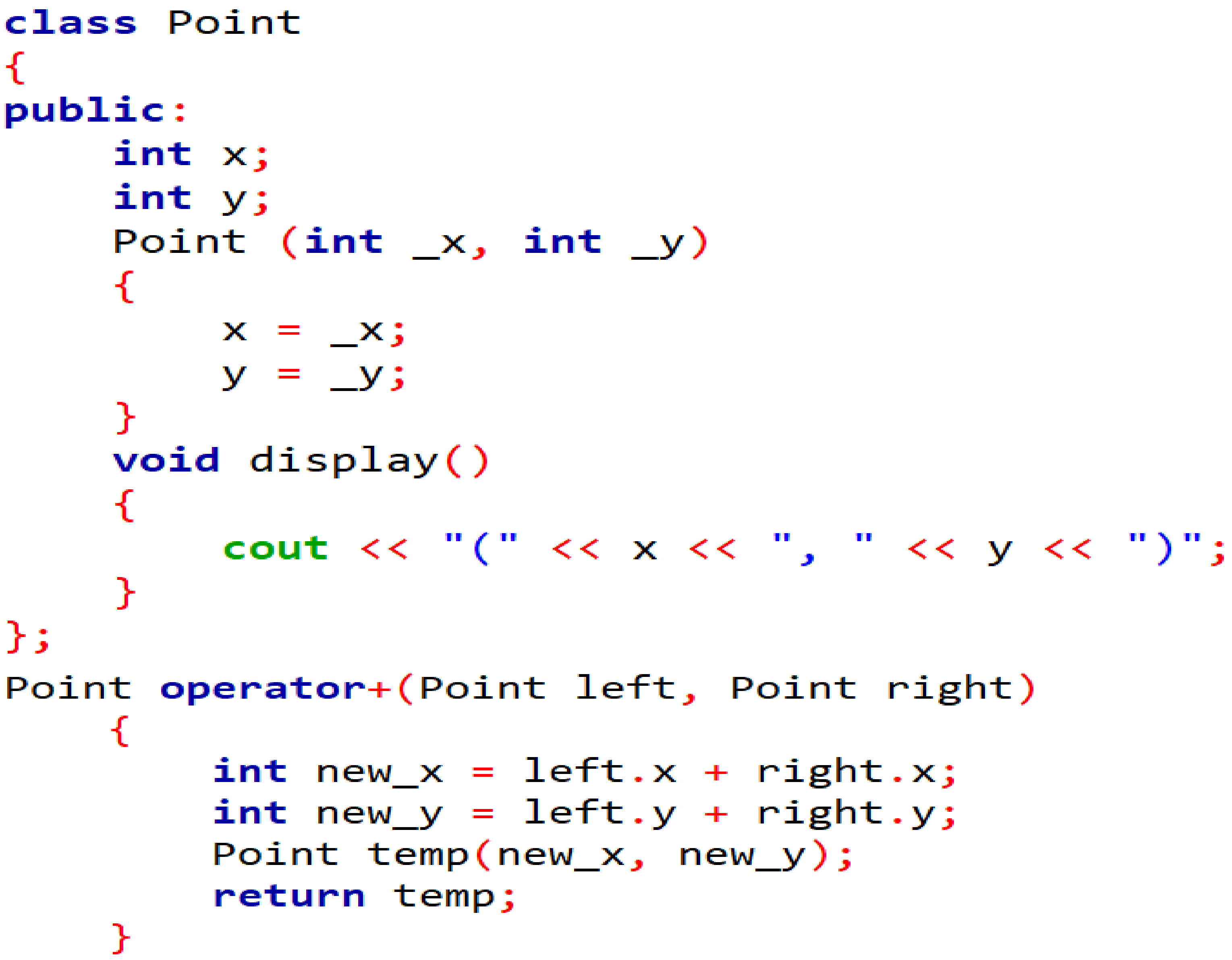


```
class Point
{
public:
    int x;
    int y;
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << "(" << x << ", " << y << ")";
    }
};
```

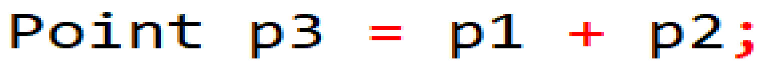


```
Point p3 = p1 + p2;
```

Operator Overloading



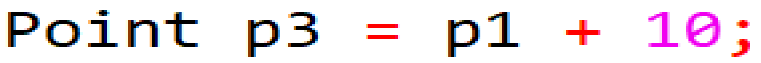
```
class Point
{
public:
    int x;
    int y;
    Point (int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << "(" << x << ", " << y << ")";
    }
};
Point operator+(Point left, Point right)
    {
        int new_x = left.x + right.x;
        int new_y = left.y + right.y;
        Point temp(new_x, new_y);
        return temp;
    }
```



```
Point p3 = p1 + p2;
```

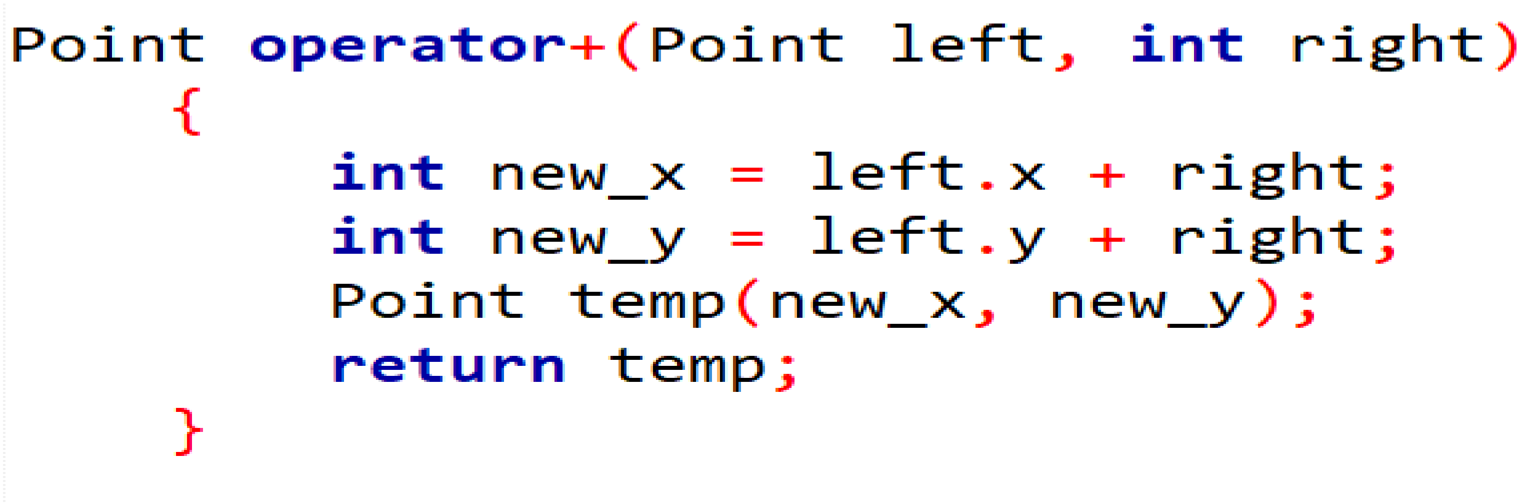
Operator Overloading

What if we wanted this?

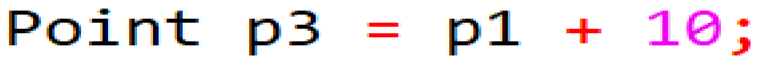


```
Point p3 = p1 + 10;
```

Operator Overloading



```
Point operator+(Point left, int right)
    {
        int new_x = left.x + right;
        int new_y = left.y + right;
        Point temp(new_x, new_y);
        return temp;
    }
```



```
Point p3 = p1 + 10;
```

Now, Let’s take a closer look at different kinds of Operator Overloading

Operator Overloading

- Enables C++ operators to work with class objects.

- Done by writing an ‘operator’ function. Eg. operator+ will overload + operator.

- Default operators of any class: ‘,’ , ‘**=**‘ and ‘**&**’

Operator Overloading Restrictions

C++ Operators that can be overloaded:

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

Pointer to member operator

C++ Operators that cannot be overloaded:

::	.*	.	?:	sizeof

Operator Overloading Restrictions

- Precedence of operator cannot be changed (order of evaluation)

  (p1 + (p2 / p3)) will not be ((p1 + p2) / p3)

- Associativity of an operator cannot be changed (left-to-right)

  ((A + B) + C) cannot be changed into (A + (B + C))

- Number of operands cannot be changed
  - Unary operator remains unary, binary operator remains binary
  - Default parameter cannot be passed
- New operator can not be created
- No overloading of built in type
  - Cannot change how two integers are added (Will produce syntax error)

Operator Overloading Placement

- Operator function as Member vs. Non-member function:

  Any operator can be non-member function **except**:

()	[]	->	Any assignment op

- Operator function as Member function:

  Leftmost operand must be an object

  (If leftmost operand is different, should make it non-member)

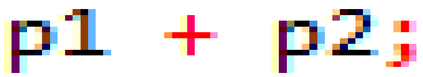
- Operator function as Non-member function:

  Must be friend of the class if private member access is required

Bottom Line: Consider making it a member function if you're dealing with private attributes.

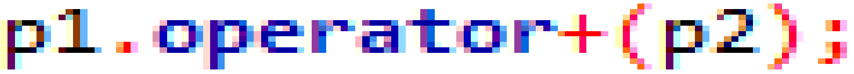
Arithmetic Operator Overloading

When we write this:



```
p1 + p2;
```

Compiler executes this:



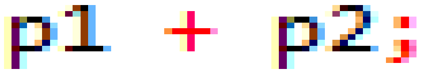
```
p1.operator+(p2);
```

This is thus a member function of

Point which we’ll have to overload

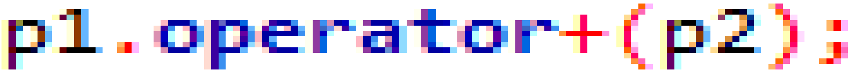
Arithmetic Operator Overloading

When we write this:



```
p1 + p2;
```

Compiler executes this:



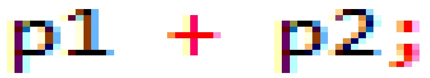
```
p1.operator+(p2);
```

The coordinates of p1 will come

from the member variable

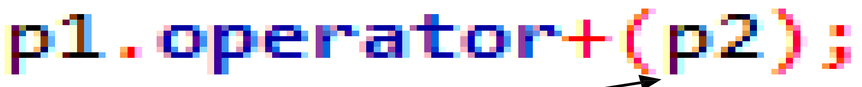
Arithmetic Operator Overloading

When we write this:



```
p1 + p2;
```

Compiler executes this:

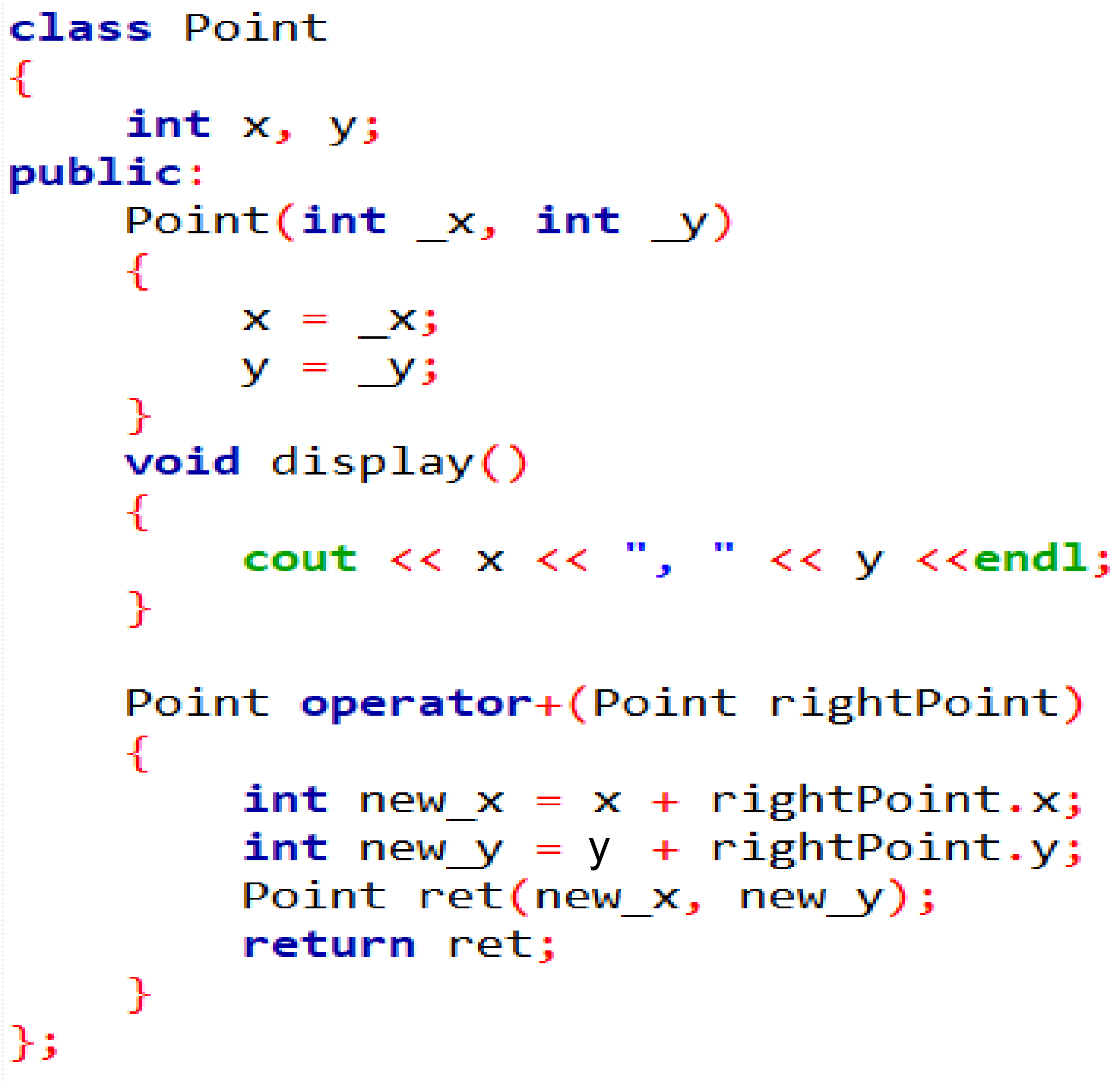


```
p1.operator+(p2);
```

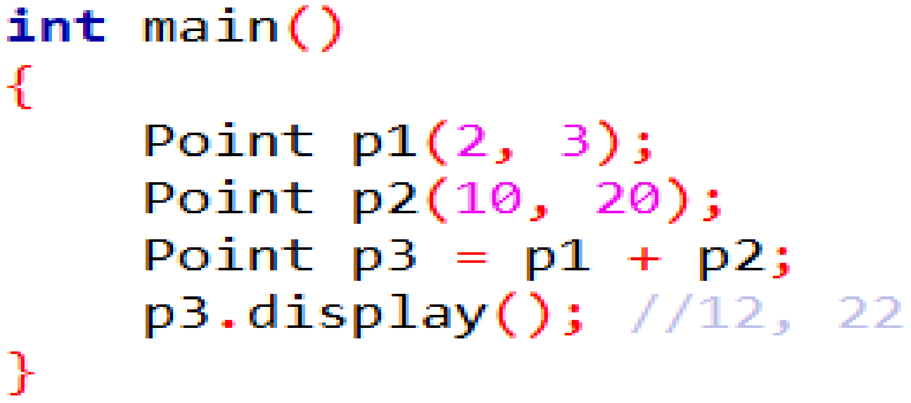
The coordinates of p2 will come

from the function argument

Arithmetic Operator Overloading



```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }

    Point operator+(Point rightPoint)
    {
        int new_x = x + rightPoint.x;
        int new_y = y + rightPoint.y;
        Point ret(new_x, new_y);
        return ret;
    }
};
```



```
int main()
{
    Point p1(2, 3);
    Point p2(10, 20);
    Point p3 = p1 + p2;
    p3.display(); //12, 22
}
```

Similar Arithmetic Operators

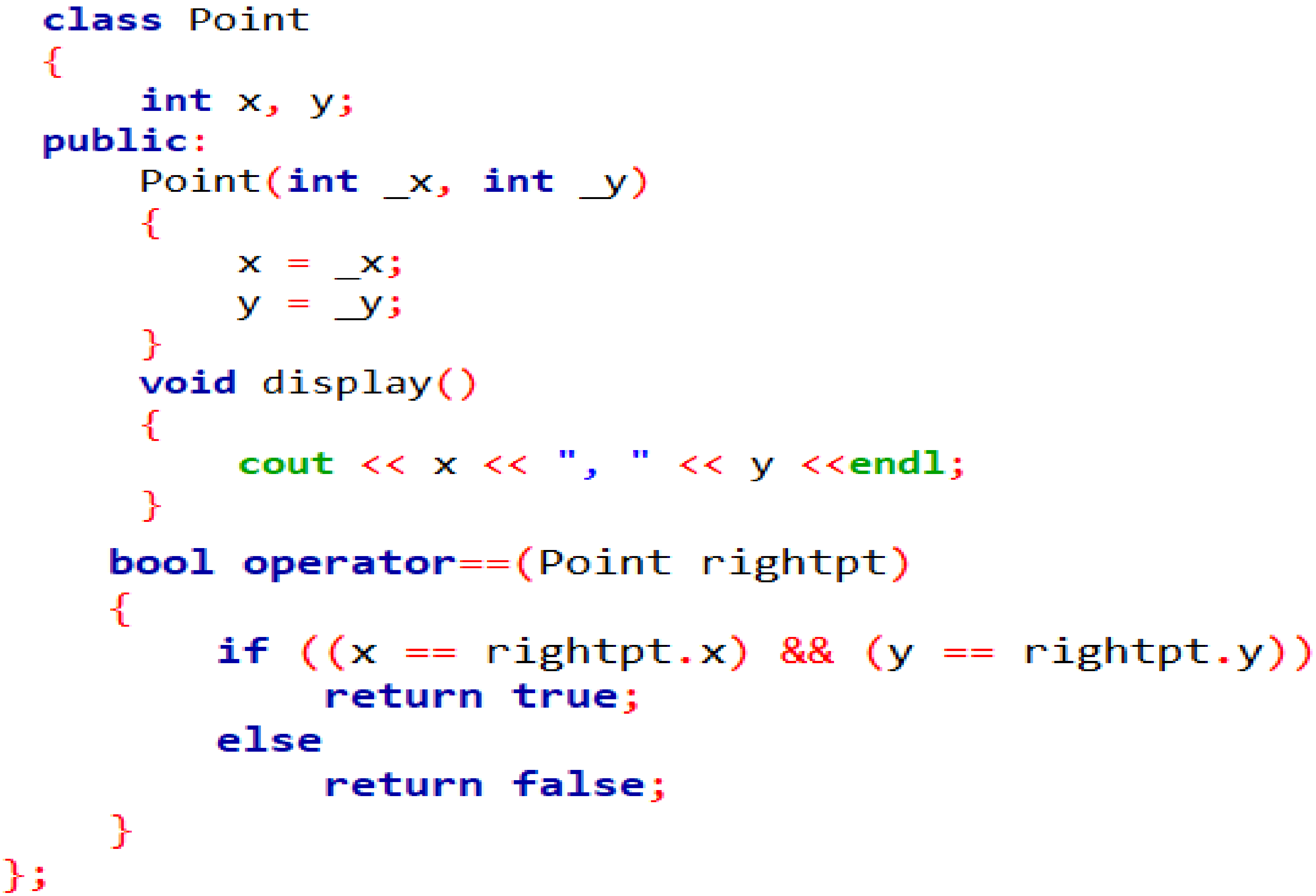
+	-	*	/	%

Relational Operators

==	!=	>	<	>=	<=

- Must return a bool value (true/false)

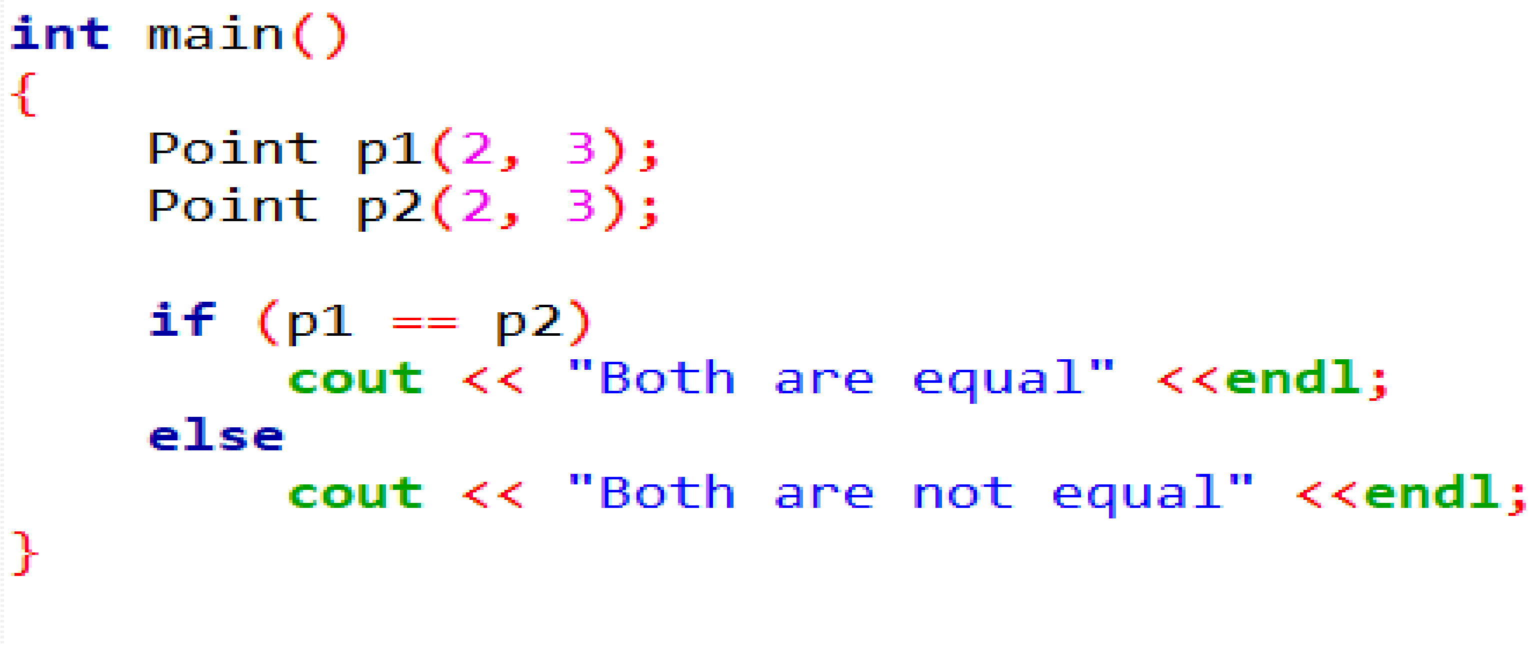
Relational Operators



```
  class Point
  {
      int x, y;
  public:
      Point(int _x, int _y)
      {
          x = _x;
          y = _y;
      }
      void display()
      {
          cout << x << ", " << y <<endl;
      }
    bool operator==(Point rightpt)
    {
         if ((x == rightpt.x) && (y == rightpt.y))
             return true;
         else
             return false;
    }
};
```

Relational Operators

…cont.



```
int main()
{
    Point p1(2, 3);
    Point p2(2, 3);

    if (p1 == p2)
        cout << "Both are equal" <<endl;
    else
        cout << "Both are not equal" <<endl;
}
```

Compound Assignment Operators

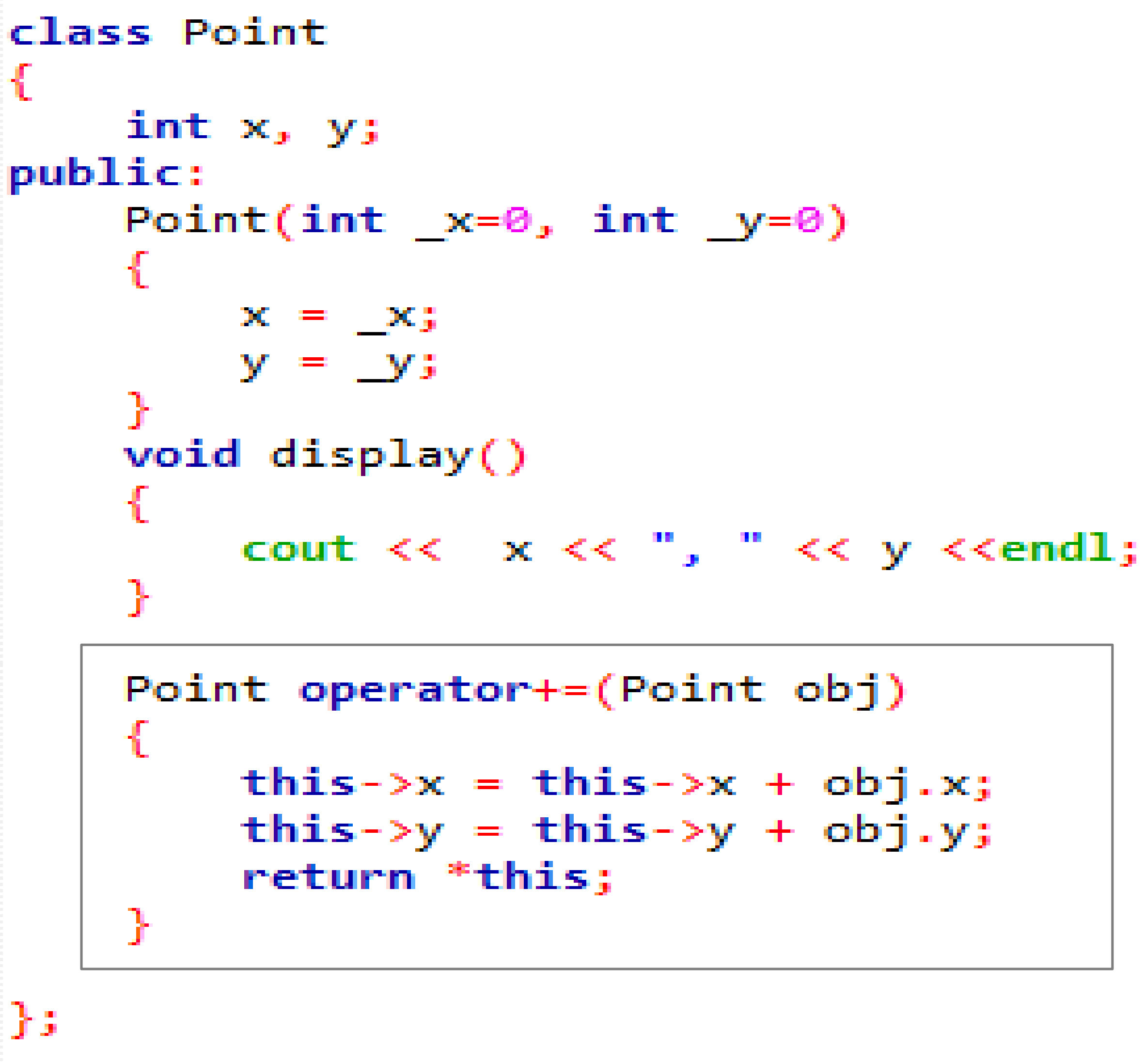
+=	-=	*=	/=	%=
&=	|=	^=	<<=	>>=

- **Changes the left hand operator**
- Should be overloaded as member function

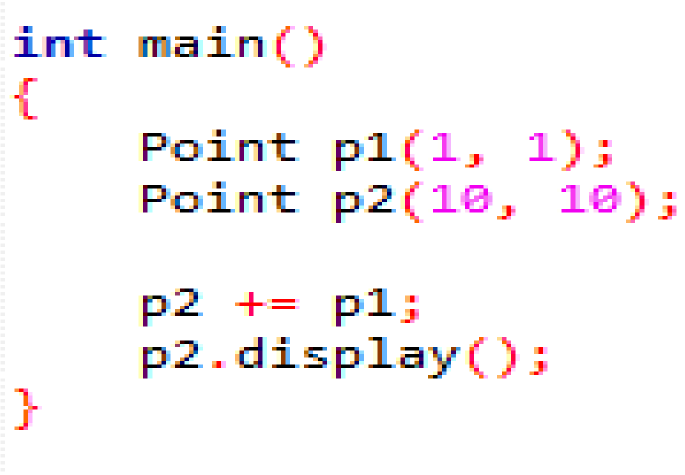
```
Point p1(1, 2), p2(10, 10);

…

p1 += p2; //p1 = (11, 12); equivalent to p1 = p1 + p2
```

Compound Assignment Operators

+= Implementation as member function



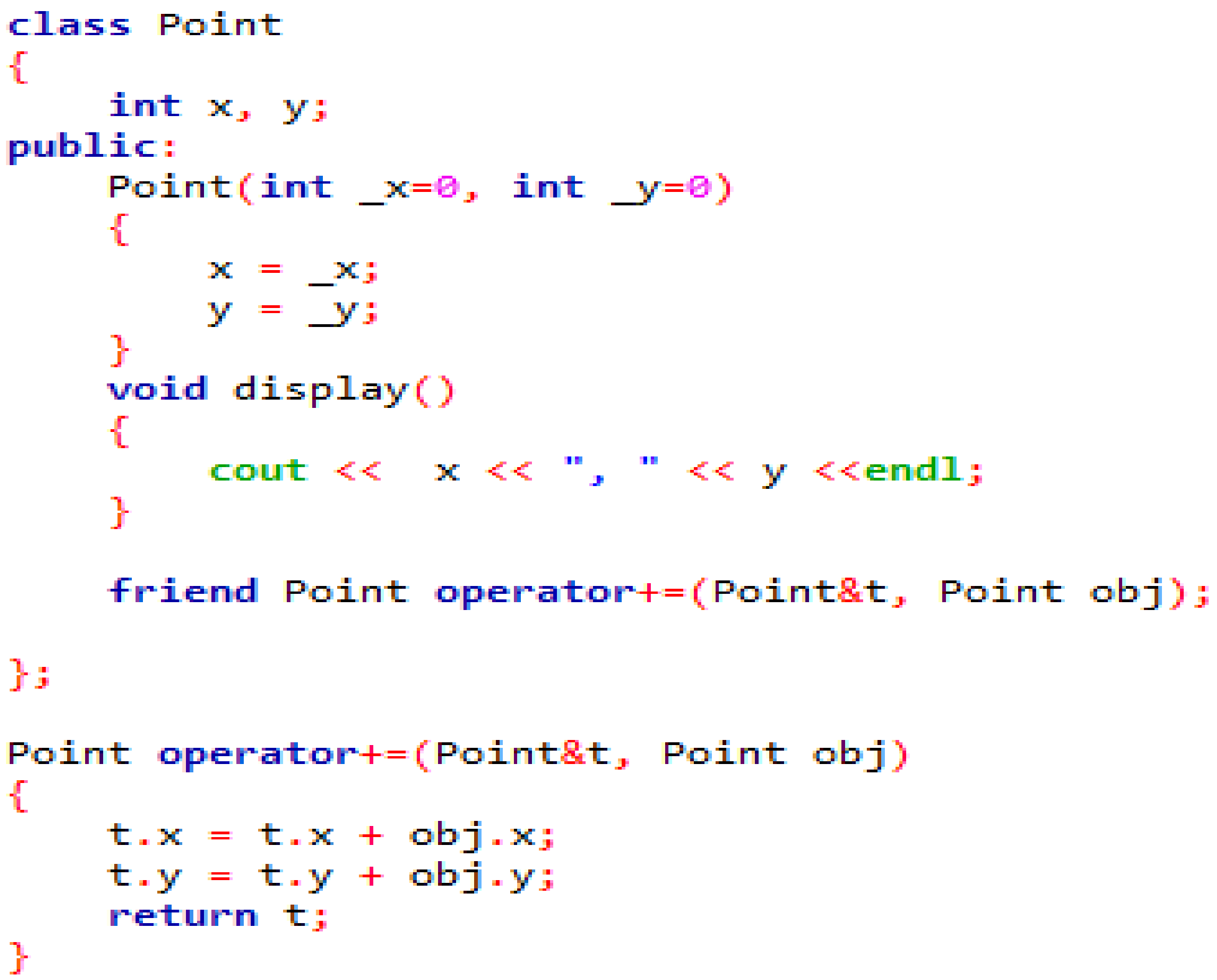
```
class Point
{
    int x, y;
public:
    Point(int _x=0, int _y=0)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout <<  x << ", " << y <<endl;
    }
    Point operator+=(Point obj)
    {
        this->x = this->x + obj.x;
        this->y = this->y + obj.y;
        return *this;
    }
};
```



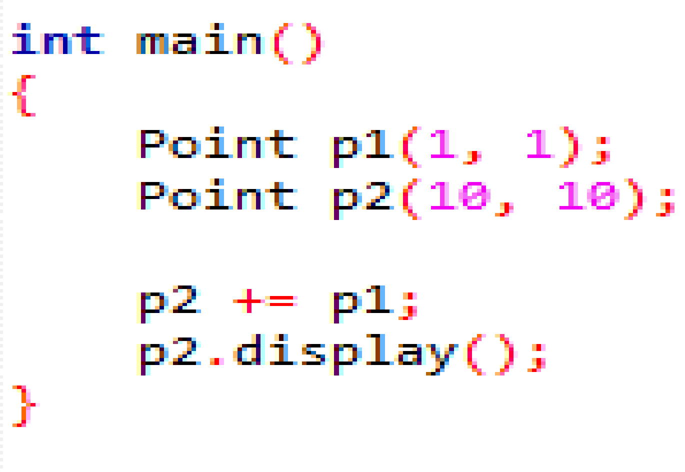
```
int main()
{
    Point p1(1, 1);
    Point p2(10, 10);

    p2 += p1;
    p2.display();
}
```

Compound Assignment Operators

+= Implementation as non-member function



```
class Point
{
    int x, y;
public:
    Point(int _x=0, int _y=0)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout <<  x << ", " << y <<endl;
    }

    friend Point operator+=(Point&t, Point obj);

};

Point operator+=(Point&t, Point obj)
{
    t.x = t.x + obj.x;
    t.y = t.y + obj.y;
    return t;
}
```

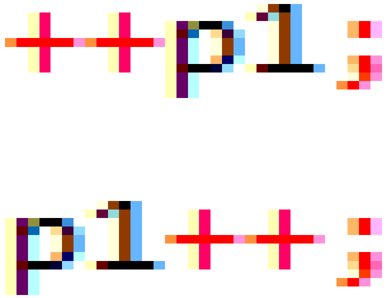


```
int main()
{
    Point p1(1, 1);
    Point p2(10, 10);

    p2 += p1;
    p2.display();
}
```

Increment/Decrement Operator

++	--

- These operators can be prefix/postfix

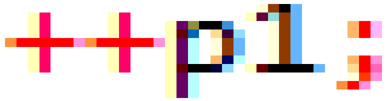


```
++p1;

p1++;
```

Increment/Decrement Operator

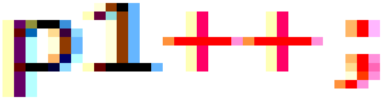
++	--

- These operators can be prefix/postfix



```
++p1;
```

Prefix



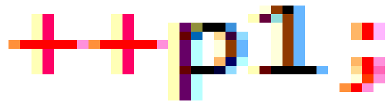
```
p1++;
```

Postfix

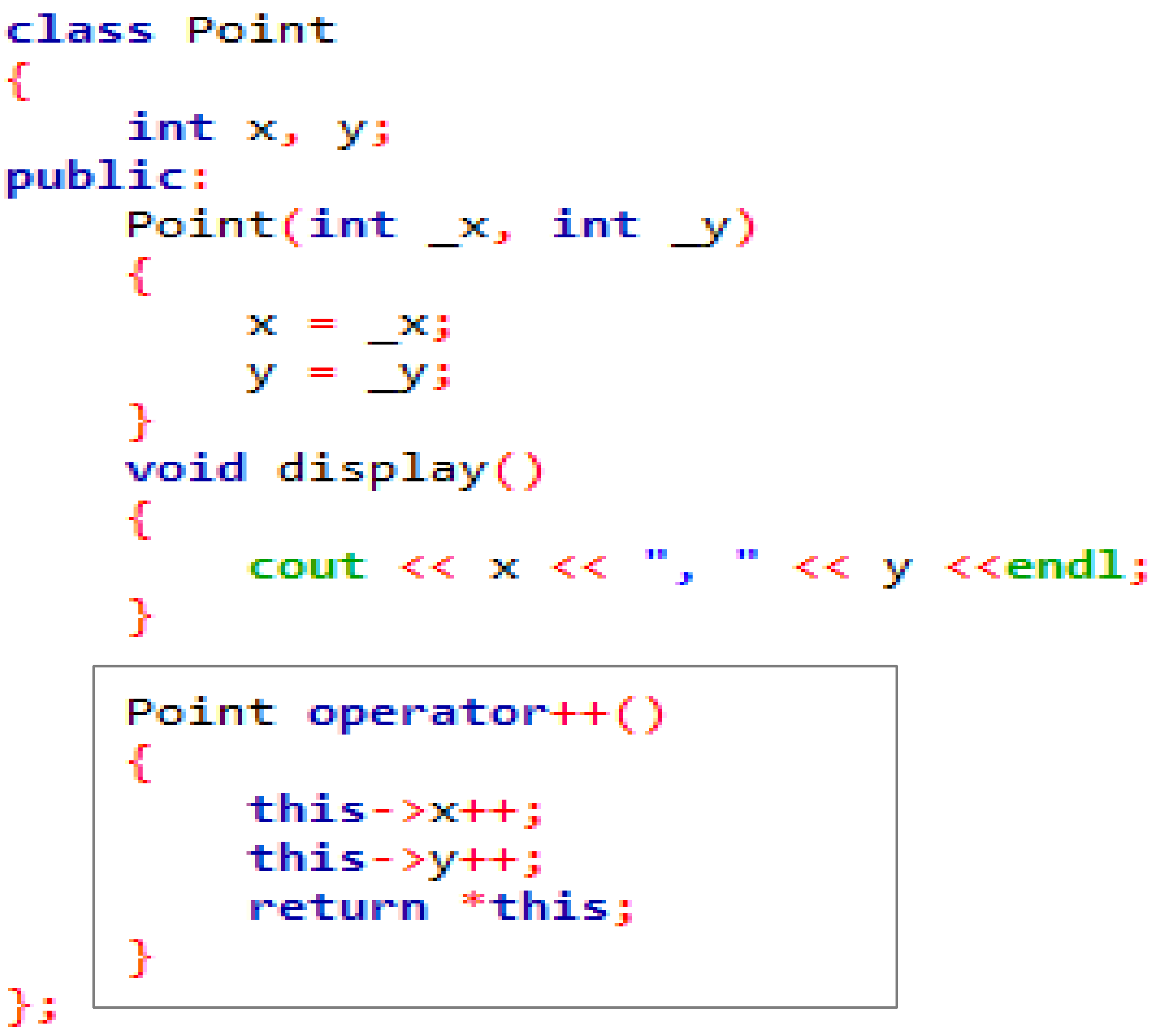
Will they return the same thing?

Prefix Increment Operator

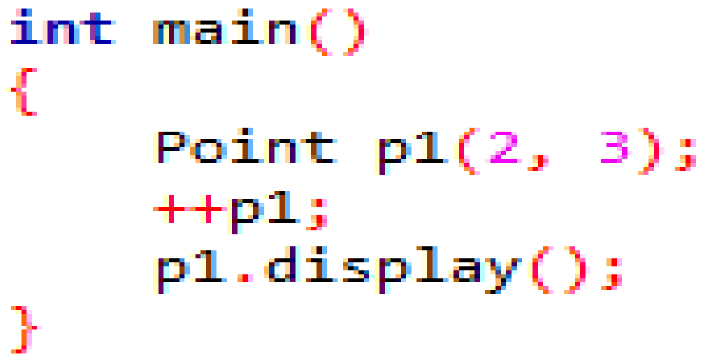
Implementation as member function



```
++p1;
```



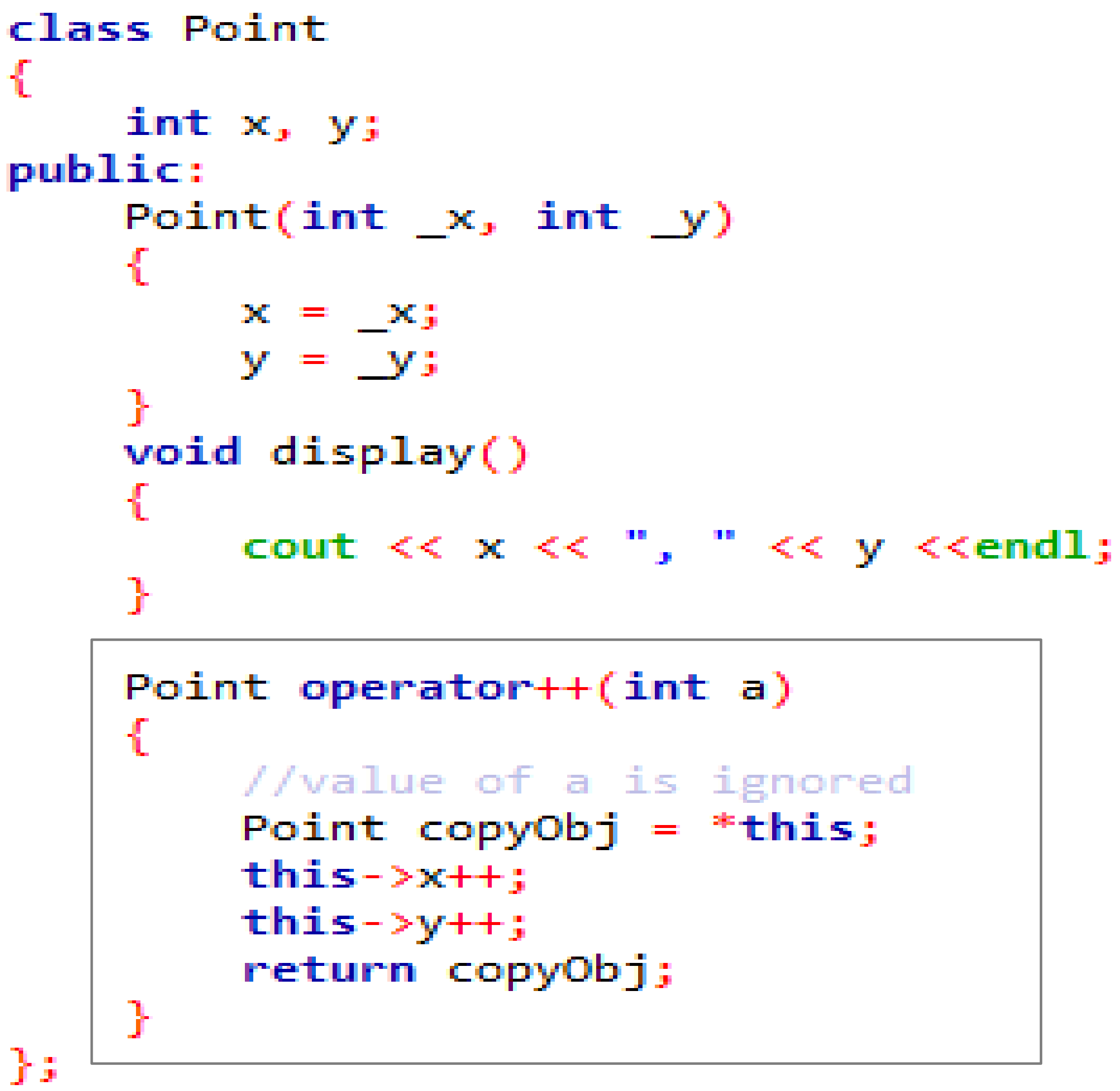
```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }
    Point operator++()
    {
        this->x++;
        this->y++;
        return *this;
    }
};
```



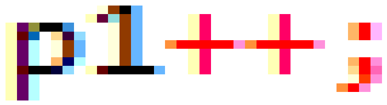
```
int main()
{
    Point p1(2, 3);
    ++p1;
    p1.display();
}
```

Postfix Increment Operator

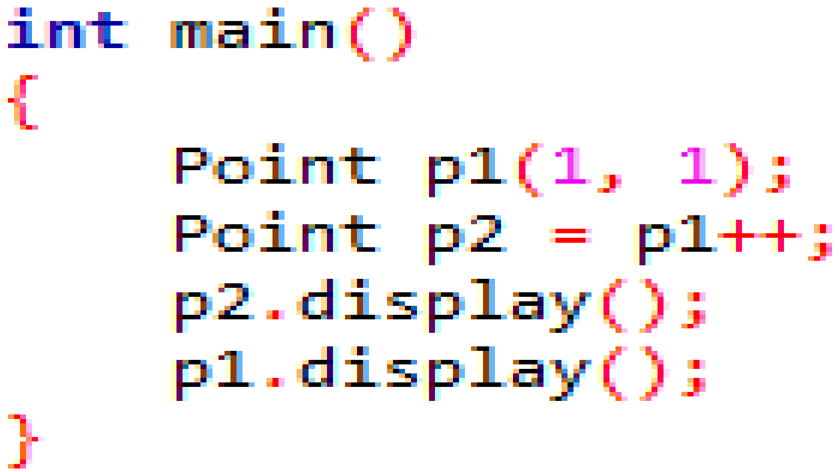
Implementation as member function



```cpp
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }
    Point operator++(int a)
    {
        //value of a is ignored
        Point copyObj = *this;
        this->x++;
        this->y++;
        return copyObj;
    }
};
```



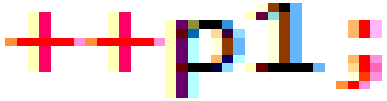
```cpp
p1++;
```



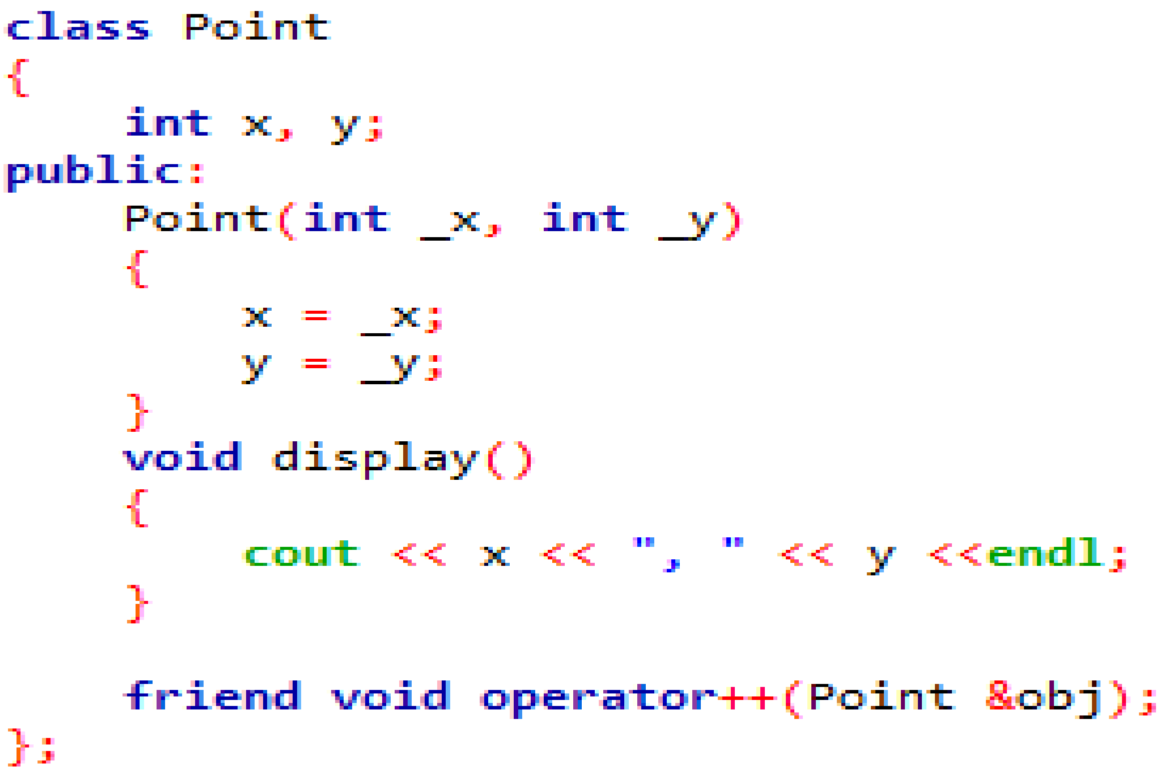
```cpp
int main()
{
    Point p1(1, 1);
    Point p2 = p1++;
    p2.display();
    p1.display();
}
```

Prefix Increment Operator

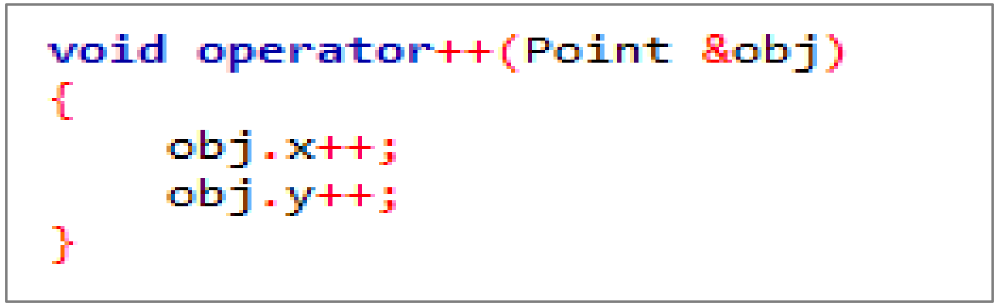
Implementation as non-member function



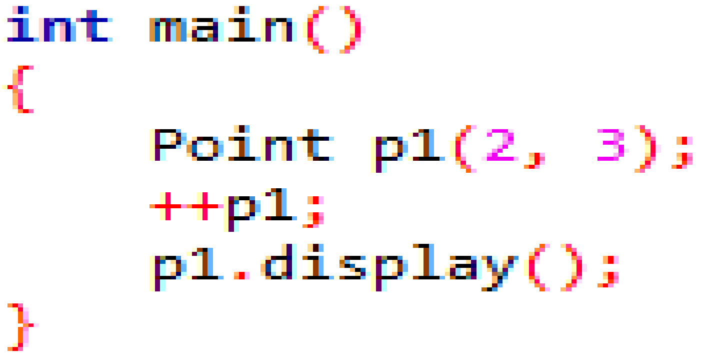
```
++p1;
```



```
class Point
{
        int x, y;
public:
        Point(int _x, int _y)
        {
                x = _x;
                y = _y;
        }
        void display()
        {
                cout << x << ", " << y <<endl;
        }

        friend void operator++(Point &obj);
};
```



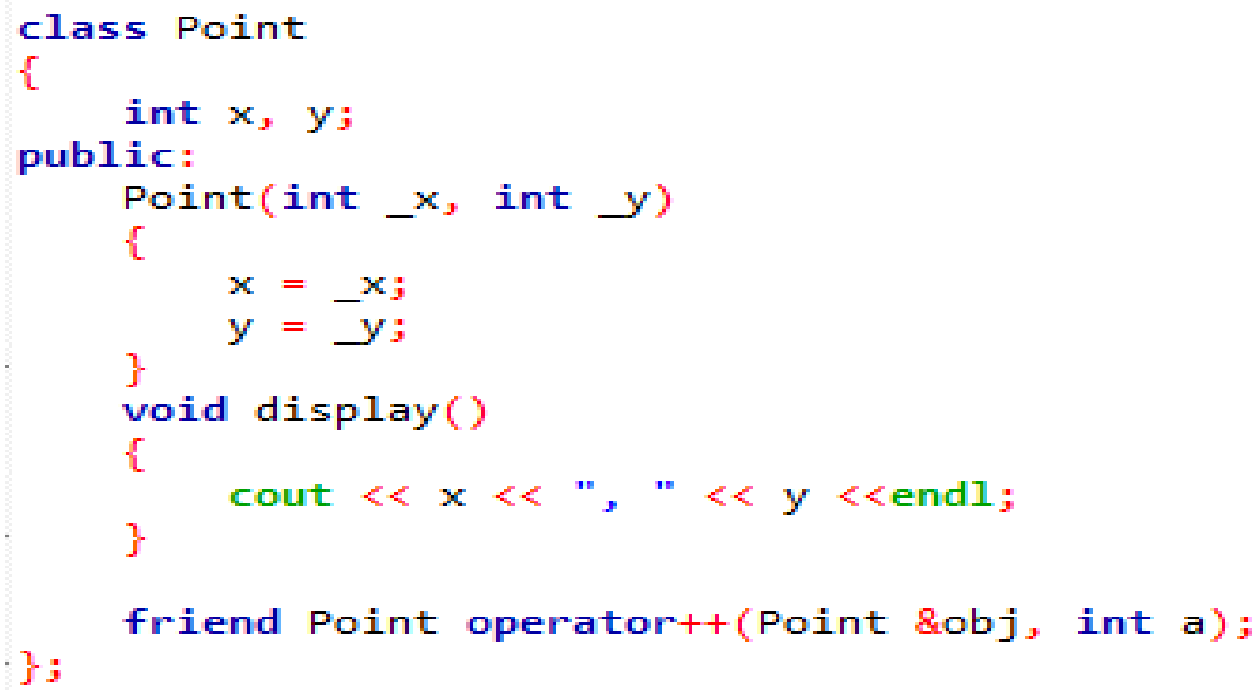
```
void operator++(Point &obj)
{
        obj.x++;
        obj.y++;
}
```



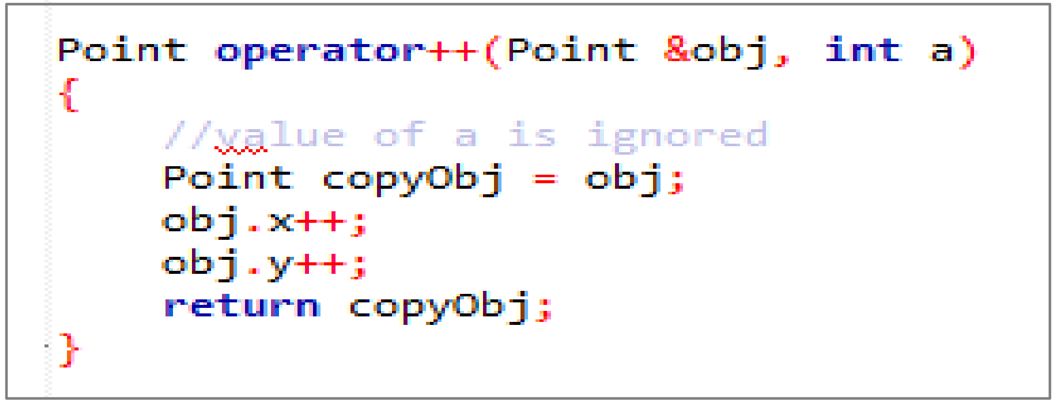
```
int main()
{
        Point p1(2, 3);
        ++p1;
        p1.display();
}
```

Postfix Increment Operator

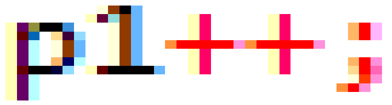
Implementation as non-member function



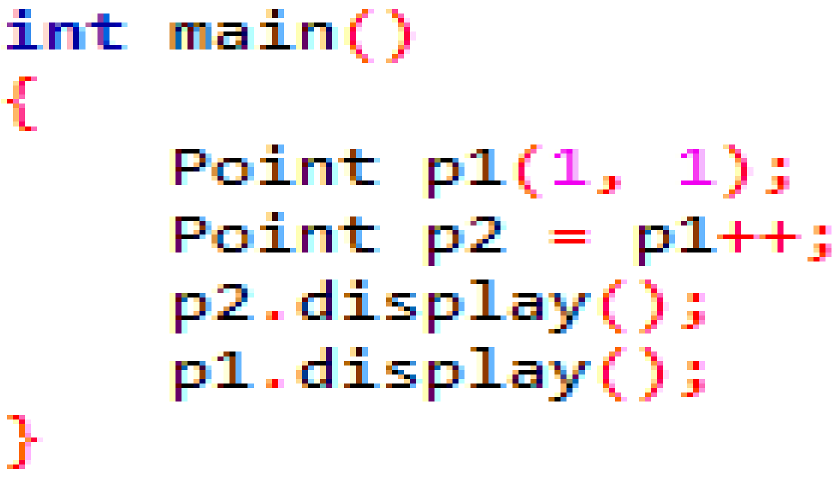
```
class Point
{
    int x, y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void display()
    {
        cout << x << ", " << y <<endl;
    }

    friend Point operator++(Point &obj, int a);
};
```



```
Point operator++(Point &obj, int a)
{
    //value of a is ignored
    Point copyObj = obj;
    obj.x++;
    obj.y++;
    return copyObj;
}
```



```
p1++;
```

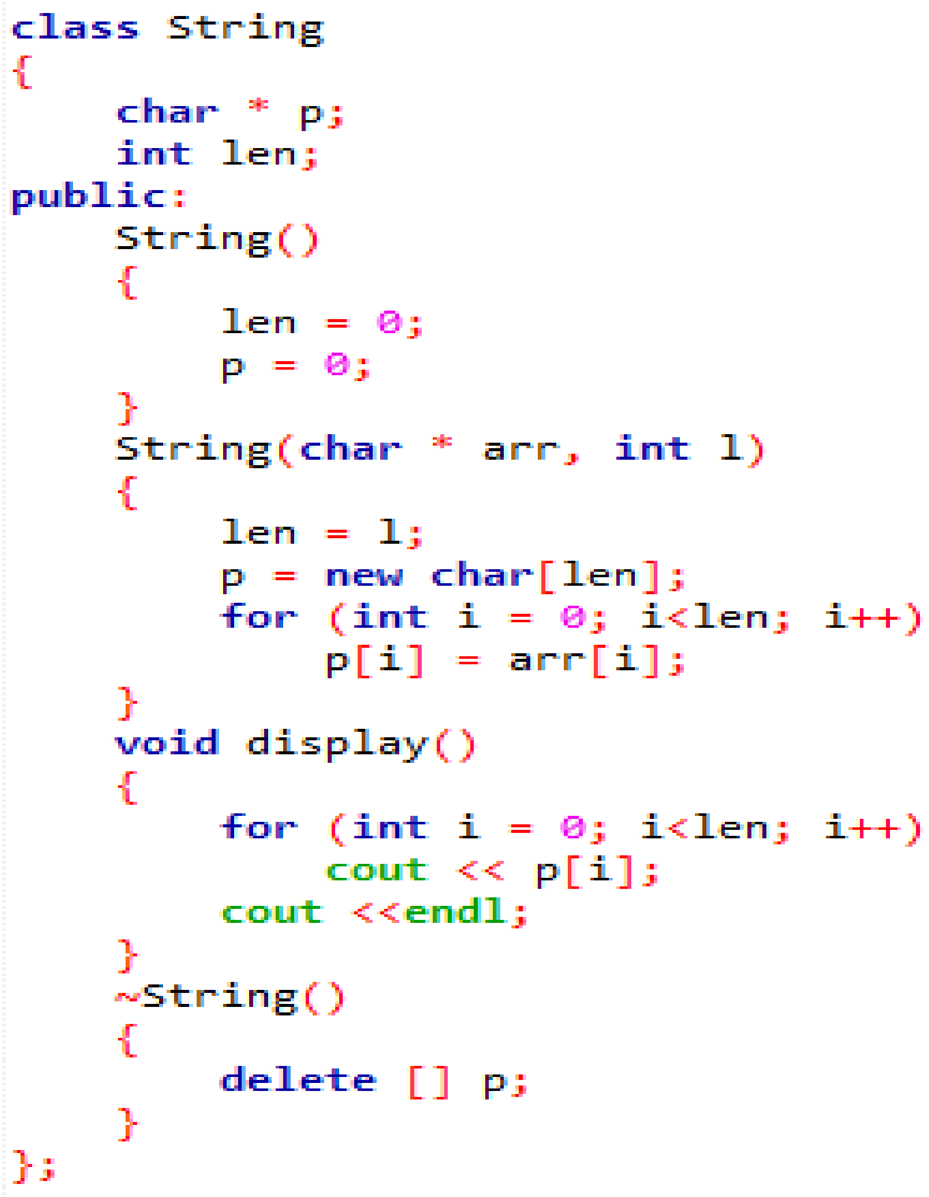


```
int main()
{
    Point p1(1, 1);
    Point p2 = p1++;
    p2.display();
    p1.display();
}
```

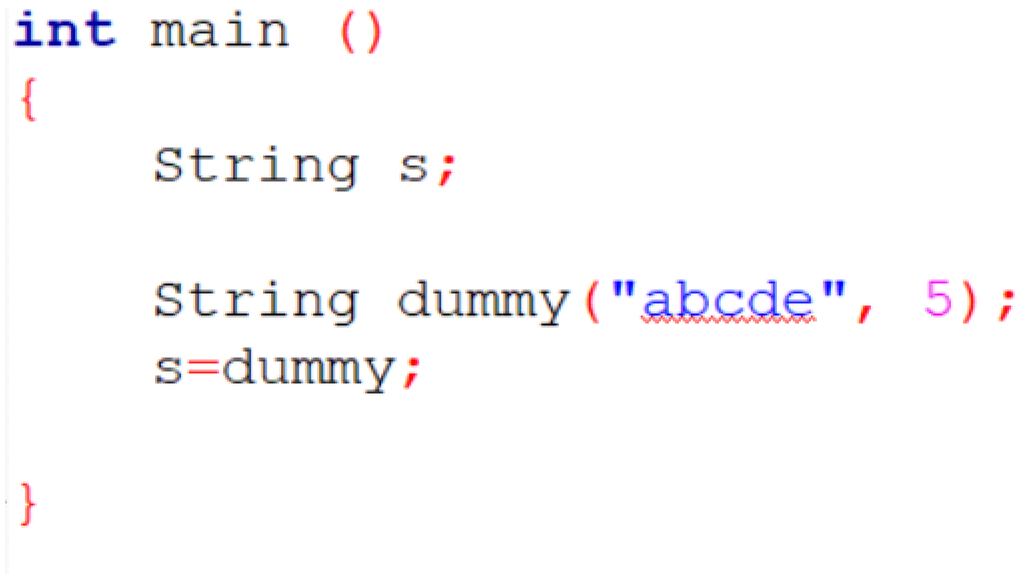
Assignment Operator =

- Must be a member function
- Receives the new value as argument, modifies this
- Should return *this to support x = y = z;

A Practical use of = overloading

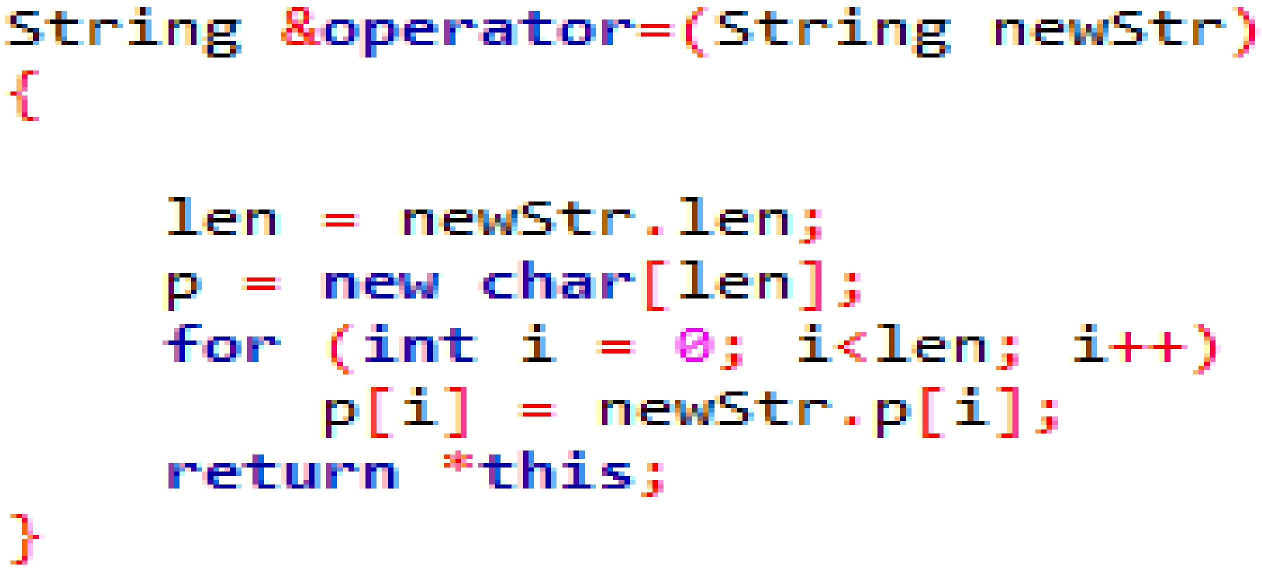


```
class String
{
    char * p;
    int len;
public:
    String()
    {
        len = 0;
        p = 0;
    }
    String(char * arr, int l)
    {
        len = l;
        p = new char[len];
        for (int i = 0; i<len; i++)
            p[i] = arr[i];
    }
    void display()
    {
        for (int i = 0; i<len; i++)
            cout << p[i];
        cout <<endl;
    }
    ~String()
    {
        delete [] p;
    }
};
```



```
int main ()
{
    String s;

    String dummy("abcde", 5);
    s=dummy;


}
```

A Practical use of = overloading



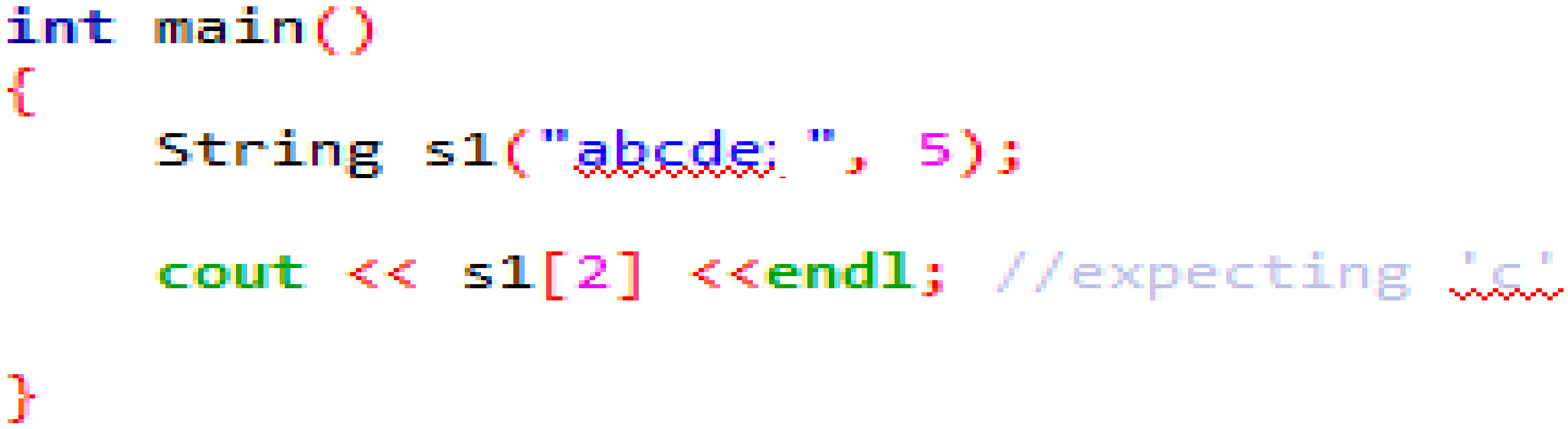
```
String &operator=(String newStr)
{

    len = newStr.len;
    p = new char[len];
    for (int i = 0; i<len; i++)
        p[i] = newStr.p[i];
    return *this;
}
```

Subscript Operator []

- **Must** be a member function
- Takes only one explicit parameter, the index
- The index can also be other than int

Subscript Operator []

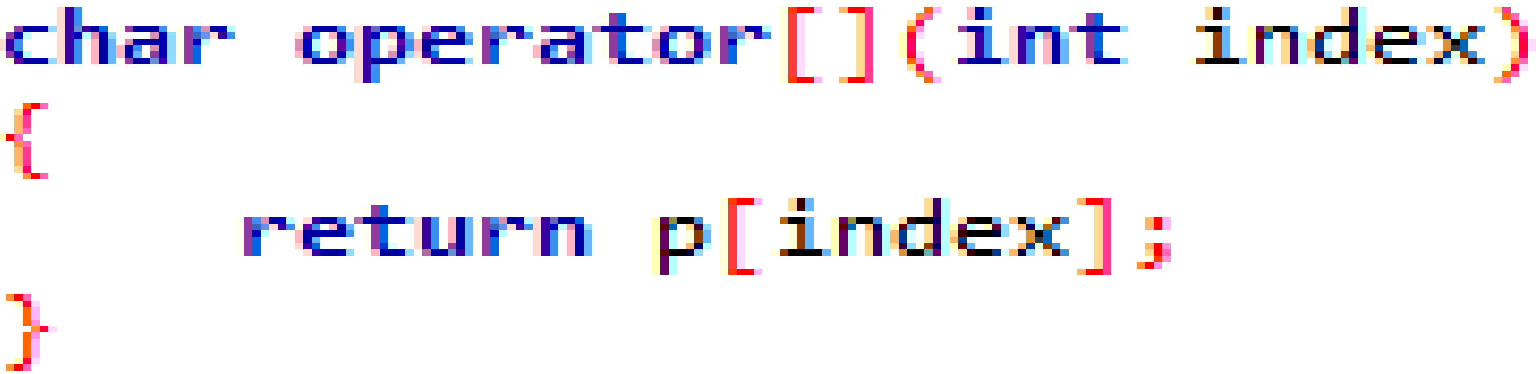
Expectation



```
int main()
{
    String s1("abcde  ", 5);

    cout << s1[2] <<endl; //expecting 'c'

}
```

Subscript Operator [] overloading

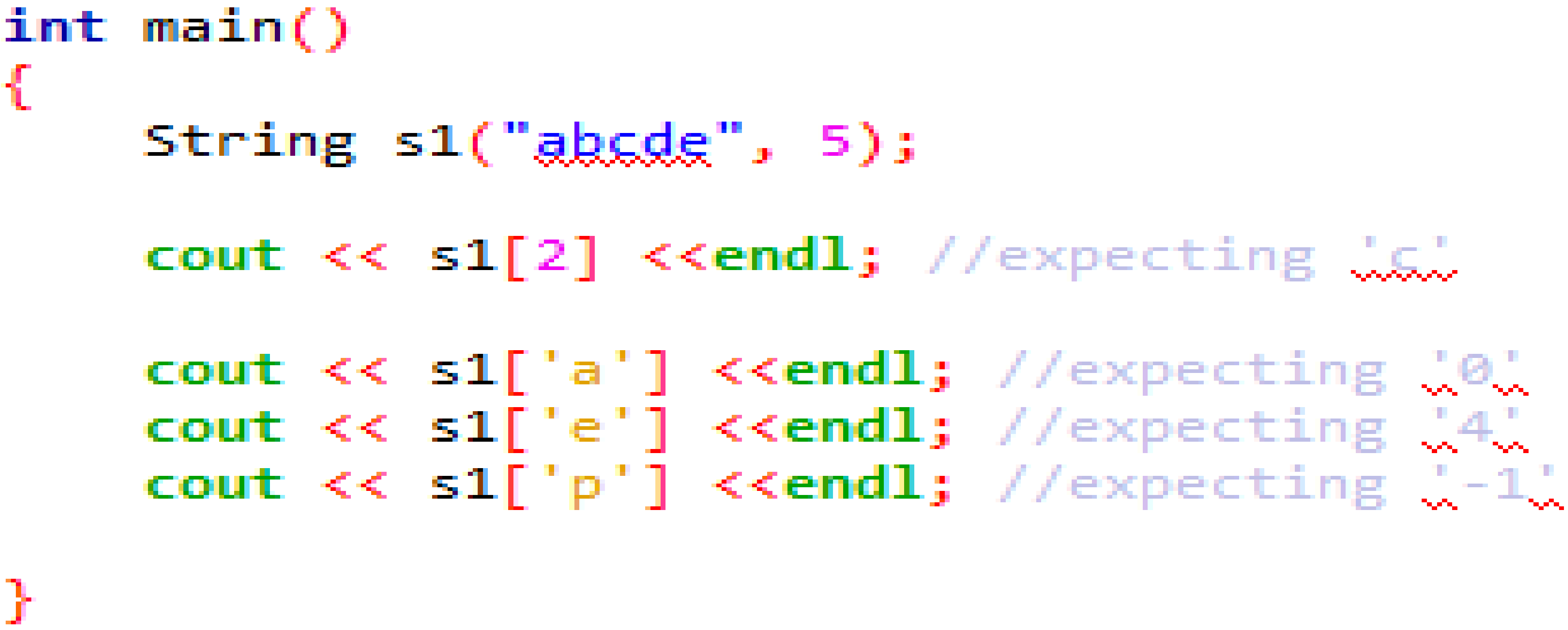
Overloaded as a member function of String



```
char operator[](int index)
{
        return p[index];
}
```

Subscript Operator [] overloading

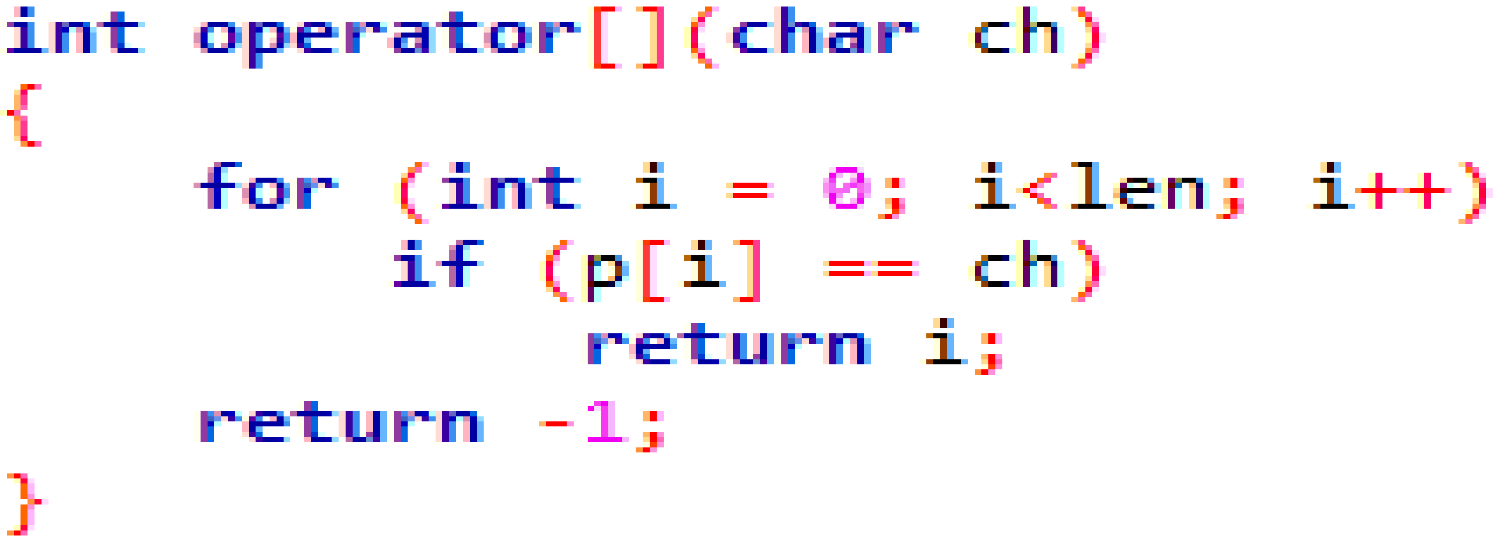
Different type of index



```
int main()
{
    String s1("abcde", 5);

    cout << s1[2] <<endl; //expecting 'c'

    cout << s1['a'] <<endl; //expecting '0'
    cout << s1['e'] <<endl; //expecting '4'
    cout << s1['p'] <<endl; //expecting '-1'

}
```

Subscript Operator [] overloading

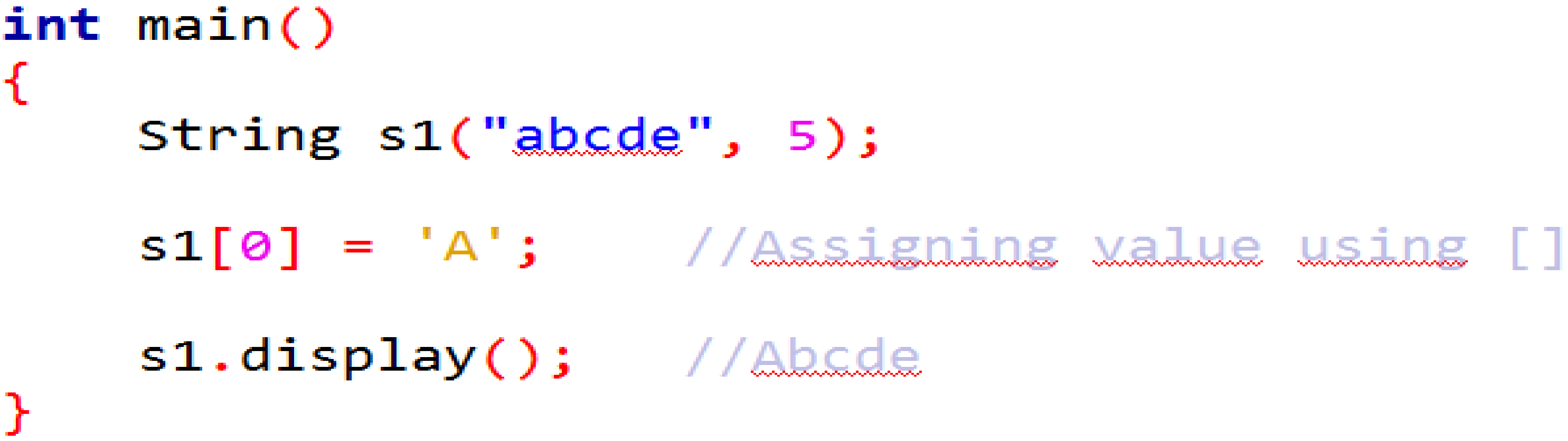
Implementation



```
int operator[](char ch)
{
    for (int i = 0; i<len; i++)
        if (p[i] == ch)
            return i;
    return -1;
}
```

Making [] work as lvalue

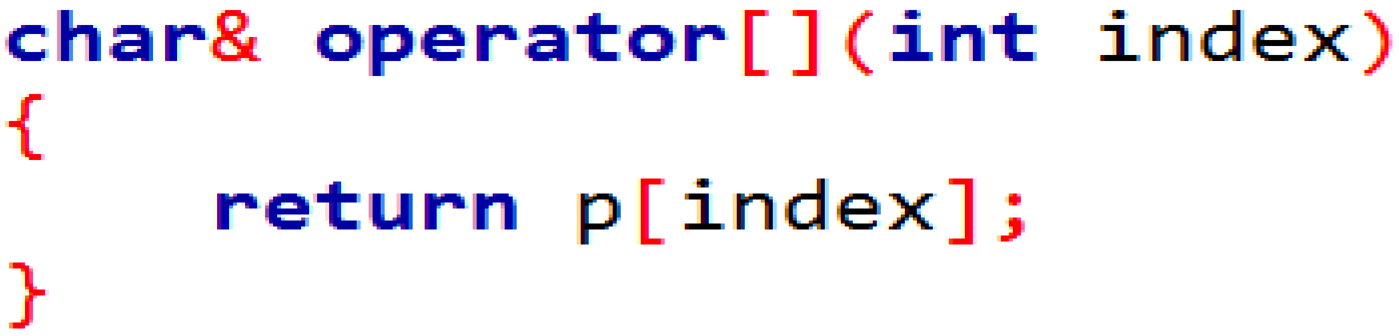
Design the operator[]() in such a way that the [ ] can be used on both the left and the right side of an assignment operator.



```
int main()
{
    String s1("abcde", 5);

    s1[0] = 'A';        //Assigning value using []

    s1.display();       //Abcde
}
```

Making [] work as lvalue

Implementation



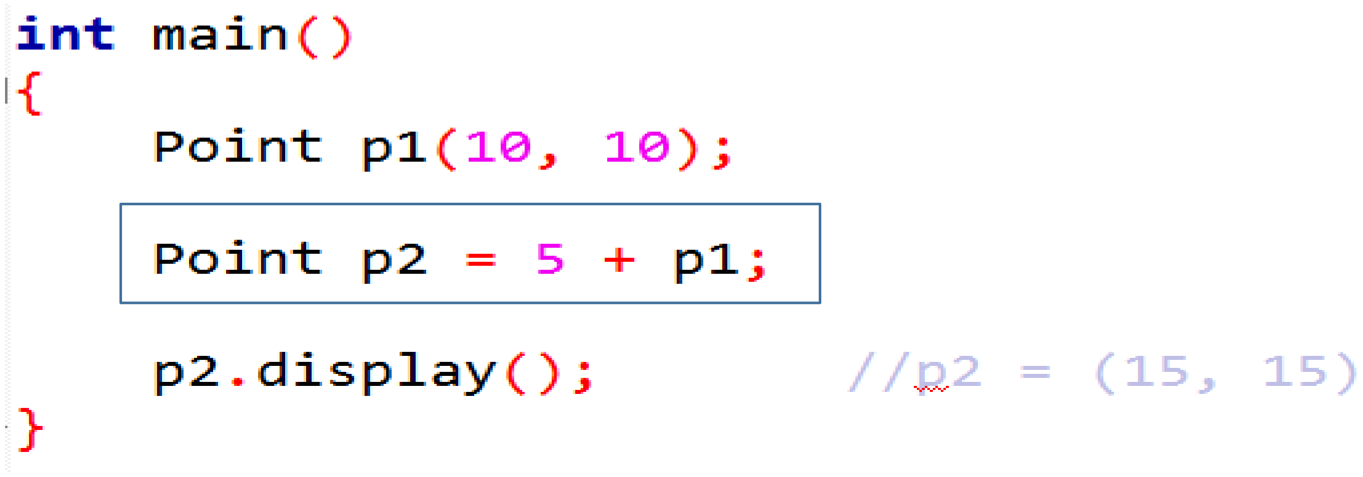
```
char& operator[](int index)
{
     return p[index];
}
```

“Now that [ ] operator returns a reference to the array element at ‘index’, It can be used on the left side of an assignment operator to modify an element of the array. Of course, it can still be used on the right side as well.”

- Teach yourself C++ by Herb Schildt (Page 225)

But What if …. ?

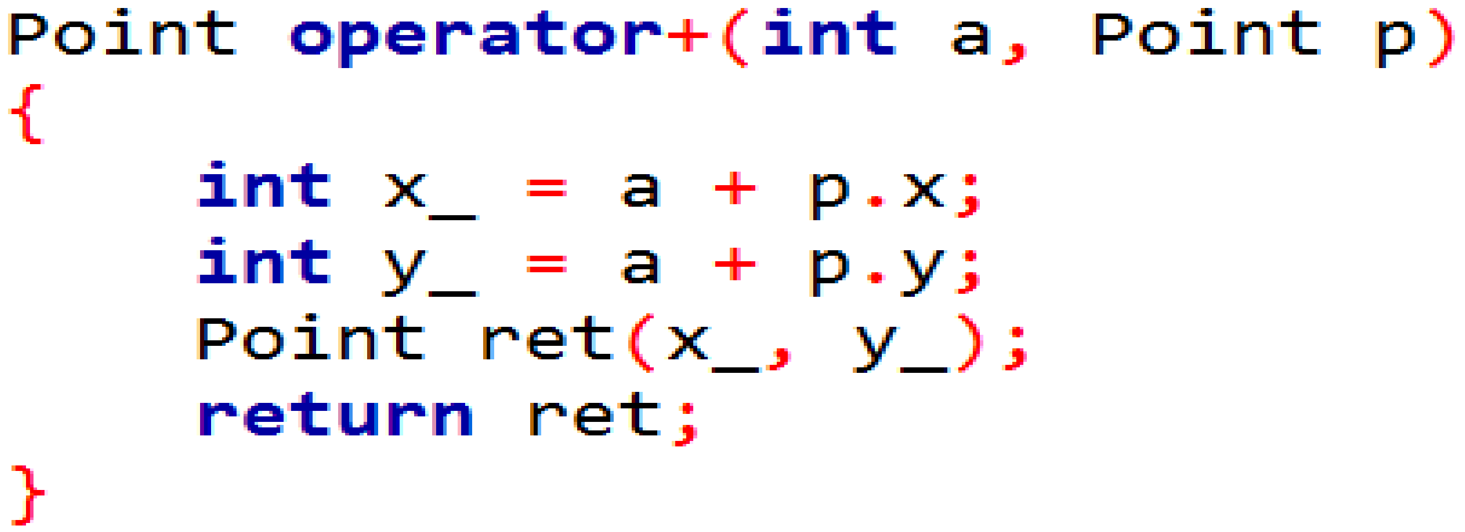
How to overload the + operator so that the following code works?



```
int main()
{
    Point p1(10, 10);

    Point p2 = 5 + p1;

    p2.display();        //p2 = (15, 15)
}
```

Solution

We must declare the operator+ function as non-member in this case.



```
Point operator+(int a, Point p)
{
    int x_ = a + p.x;
    int y_ = a + p.y;
    Point ret(x_, y_);
    return ret;
}
```

Overloading the () operator

Example:

Suppose you are to required to calculate the value of y for the following line equations, where the values of x are from 1 to 5.

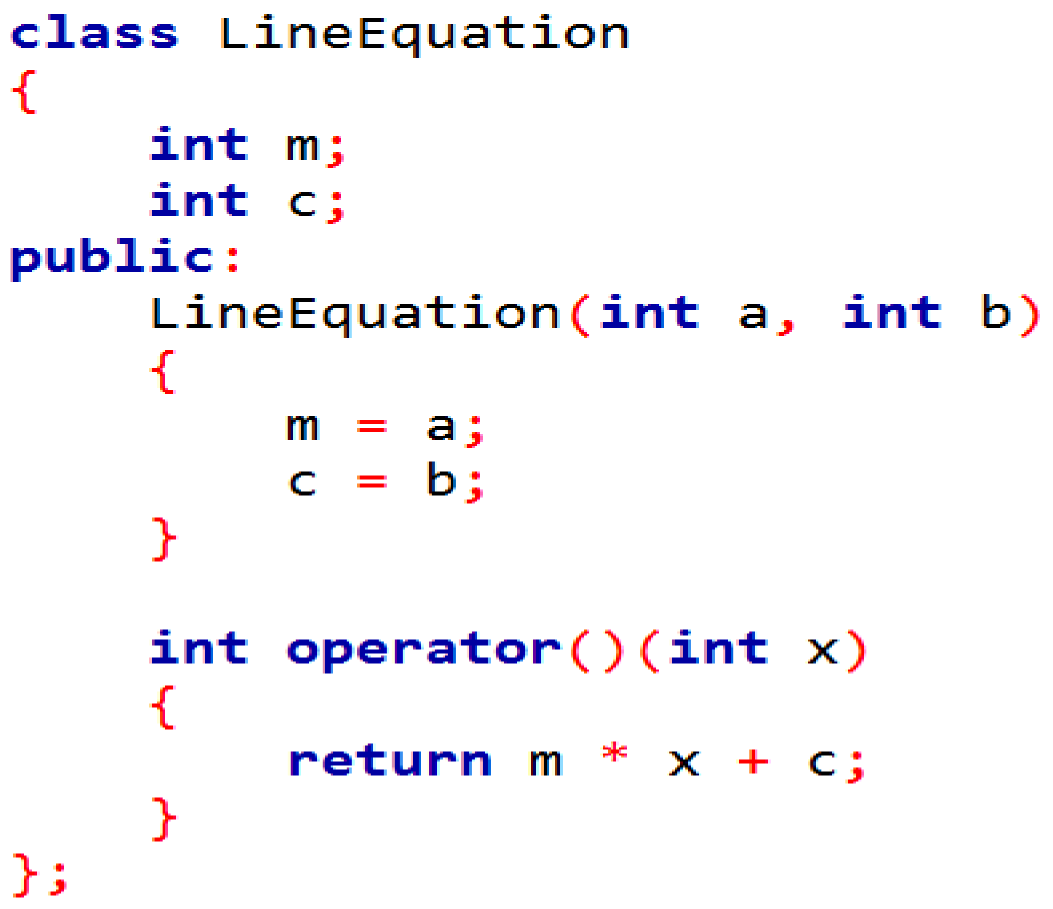
$$y = 4x + 3$$

$$y = 7x - 2$$

$$y = 2x + 5$$

Will you write three separate functions?

Overloading the () operator



```
class LineEquation
{
    int m;
    int c;
public:
    LineEquation(int a, int b)
    {
        m = a;
        c = b;
    }

    int operator()(int x)
    {
        return m * x + c;
    }
};
```

$y = 4x + 3$

$y = 7x - 2$

$y = 2x + 5$

Overloading the () operator

- Enabling the object to act like a function

  obj1(param1, param2,...)

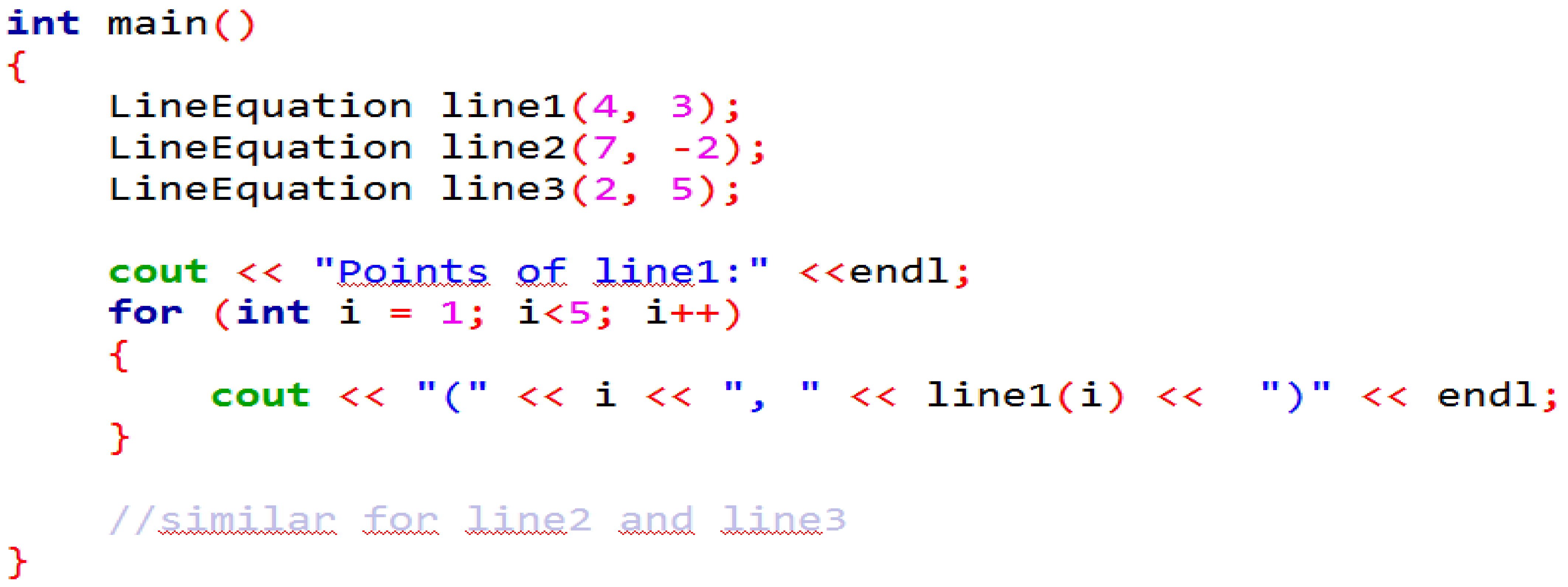
- Must be a member function of the class
- It can have any number of parameters and any return type
- The object works like a programmable function

Overloading the () operator

$$y = 4x + 3$$

$$y = 7x - 2$$

$$y = 2x + 5$$



```
int main()
{
    LineEquation line1(4, 3);
    LineEquation line2(7, -2);
    LineEquation line3(2, 5);

    cout << "Points of line1:" <<endl;
    for (int i = 1; i<5; i++)
    {
        cout << "(" << i << ", " << line1(i) <<  ")" << endl;
    }

    //similar for line2 and line3
}
```

Functor (Function Object)

Yes, you’ve read it right. Functor.

- Functor is a C++ class that acts like function.
- It’s a class where operator () is defined.
- line1, line2, line3 in the previous example are Functors.

Functor vs Function Pointer

- Functors are more efficient than Function Pointer. Function pointer may require runtime pointer dereferencing.

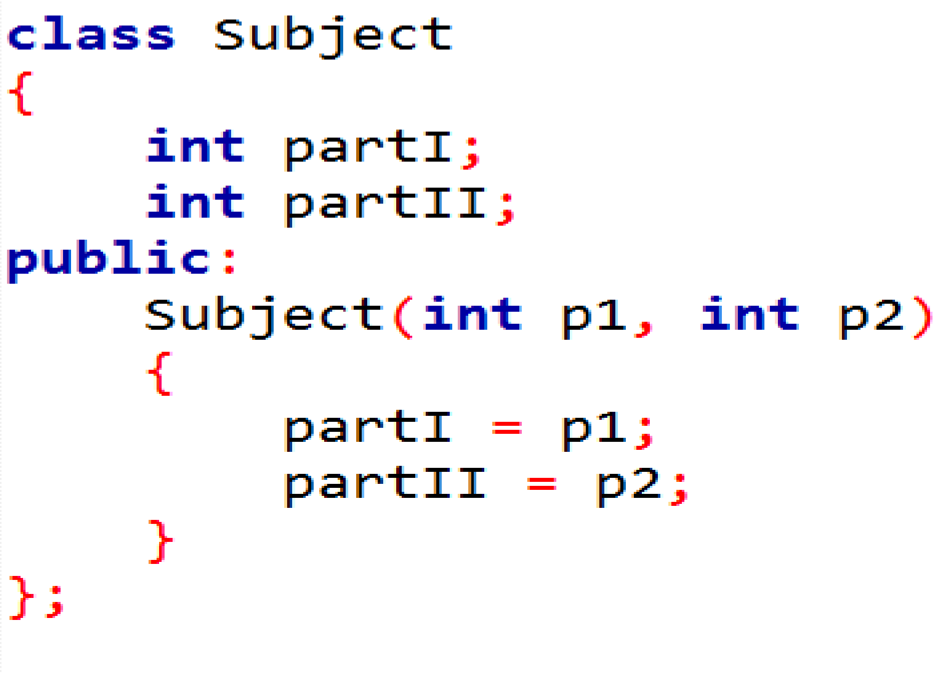
- Functor can contain state

Conversion Function

When we want to convert an object of one type to another

Conversion Function

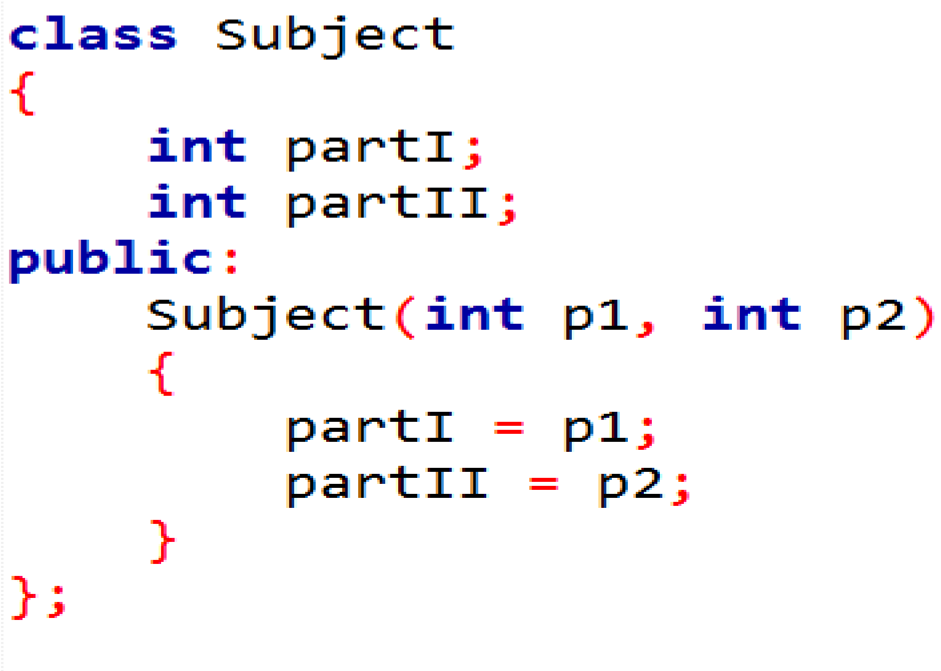
When we want to convert an object of one type to another



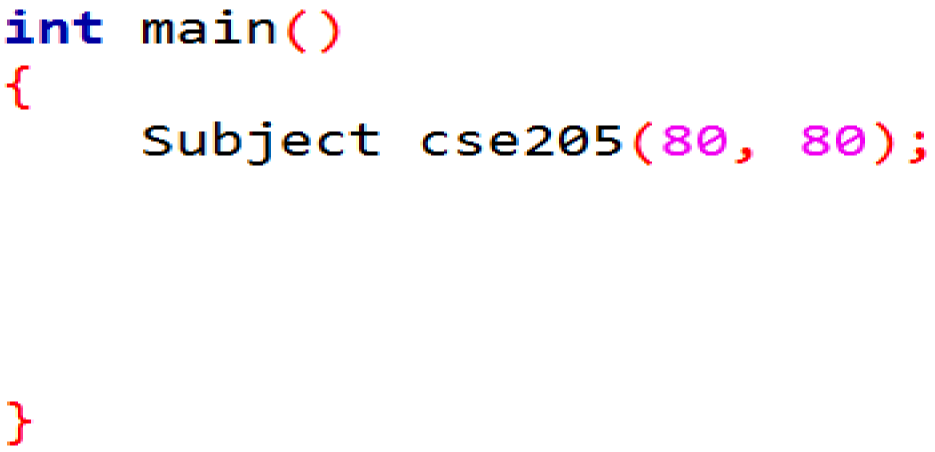
```
class Subject
{
    int partI;
    int partII;
public:
    Subject(int p1, int p2)
    {
        partI = p1;
        partII = p2;
    }
};
```

Conversion Function

When we want to convert an object of one type to another



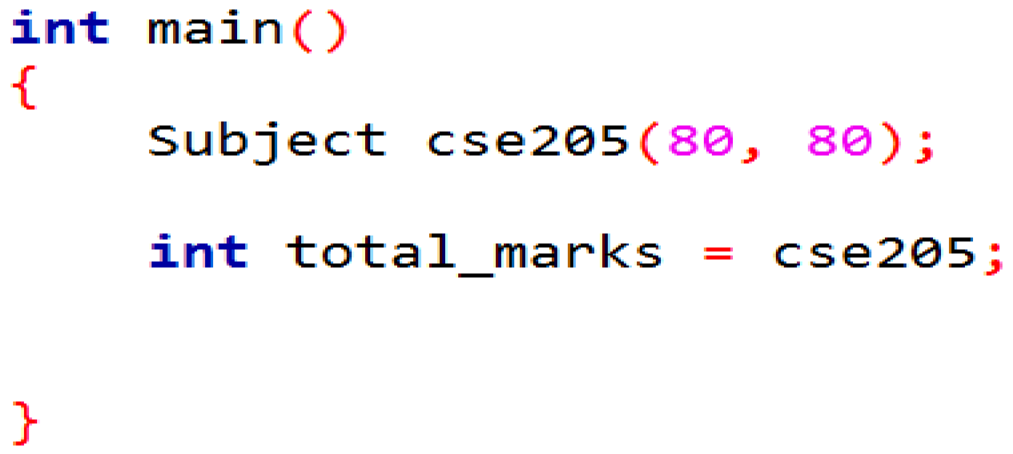
```
class Subject
{
    int partI;
    int partII;
public:
    Subject(int p1, int p2)
    {
        partI = p1;
        partII = p2;
    }
};
```



```
int main()
{
    Subject cse205(80, 80);



}
```

Conversion Function

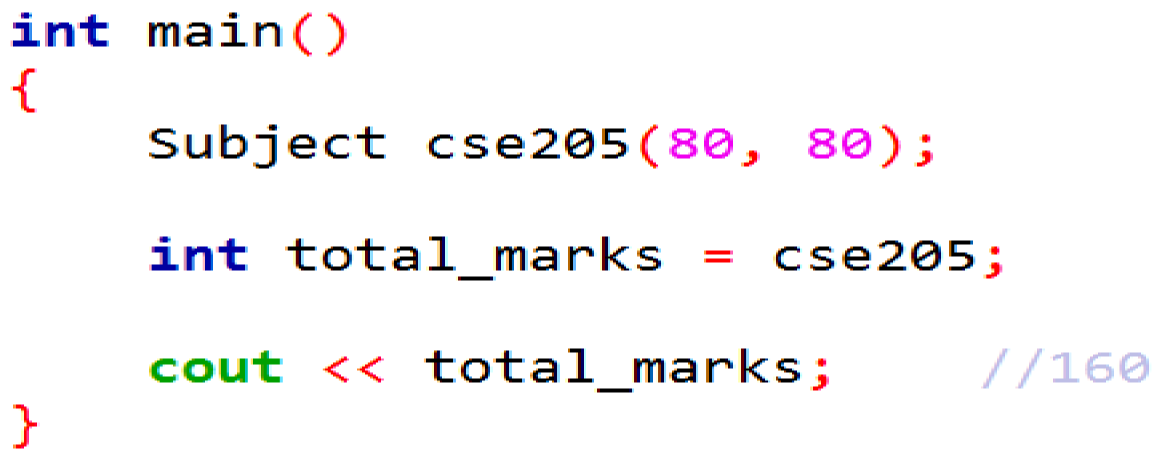
When we want to convert an object of one type to another



```
int main()
{
    Subject cse205(80, 80);

    int total_marks = cse205;


}
```

Conversion Function

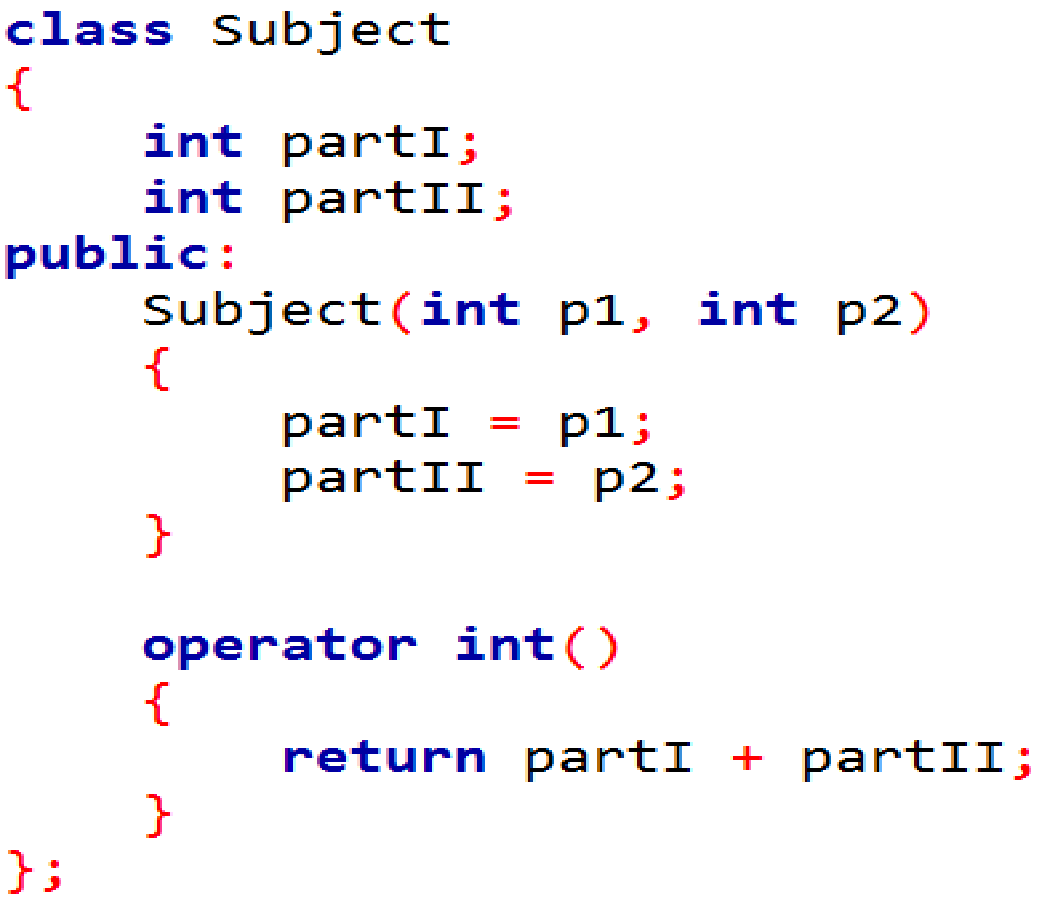
When we want to convert an object of one type to another



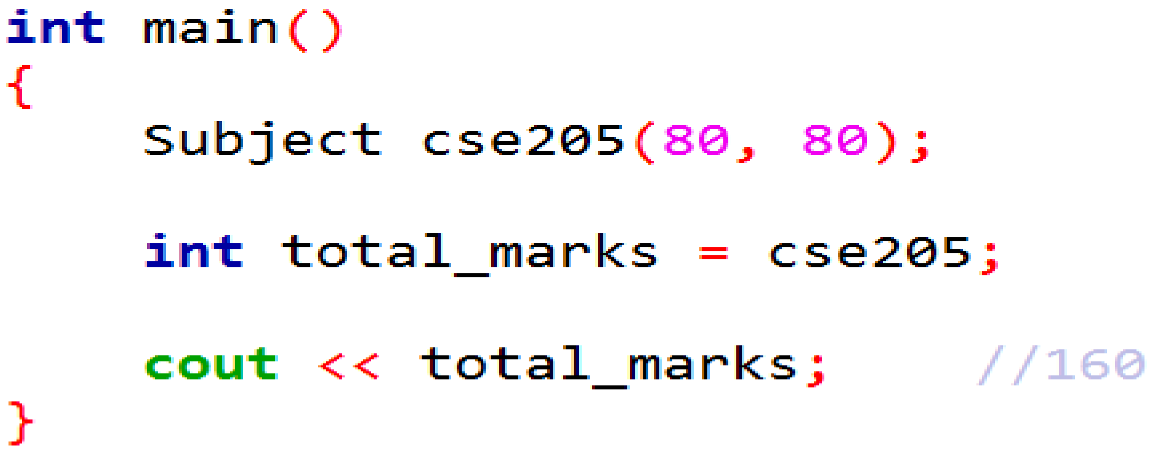
```
int main()
{
    Subject cse205(80, 80);

    int total_marks = cse205;

    cout << total_marks;    //160
}
```

Conversion Function

Syntax: operator *type*() { return value; }



```
class Subject
{
    int partI;
    int partII;
public:
    Subject(int p1, int p2)
    {
        partI = p1;
        partII = p2;
    }

    operator int()
    {
        return partI + partII;
    }
};
```



```
int main()
{
    Subject cse205(80, 80);

    int total_marks = cse205;

    cout << total_marks;     //160
}
```

Conversion Function

Syntax: operator *type*() { return value; }

- operator and return are keywords
- *type* is the target type we'll be converting our object to
- *value* is the value of the object after the conversion has

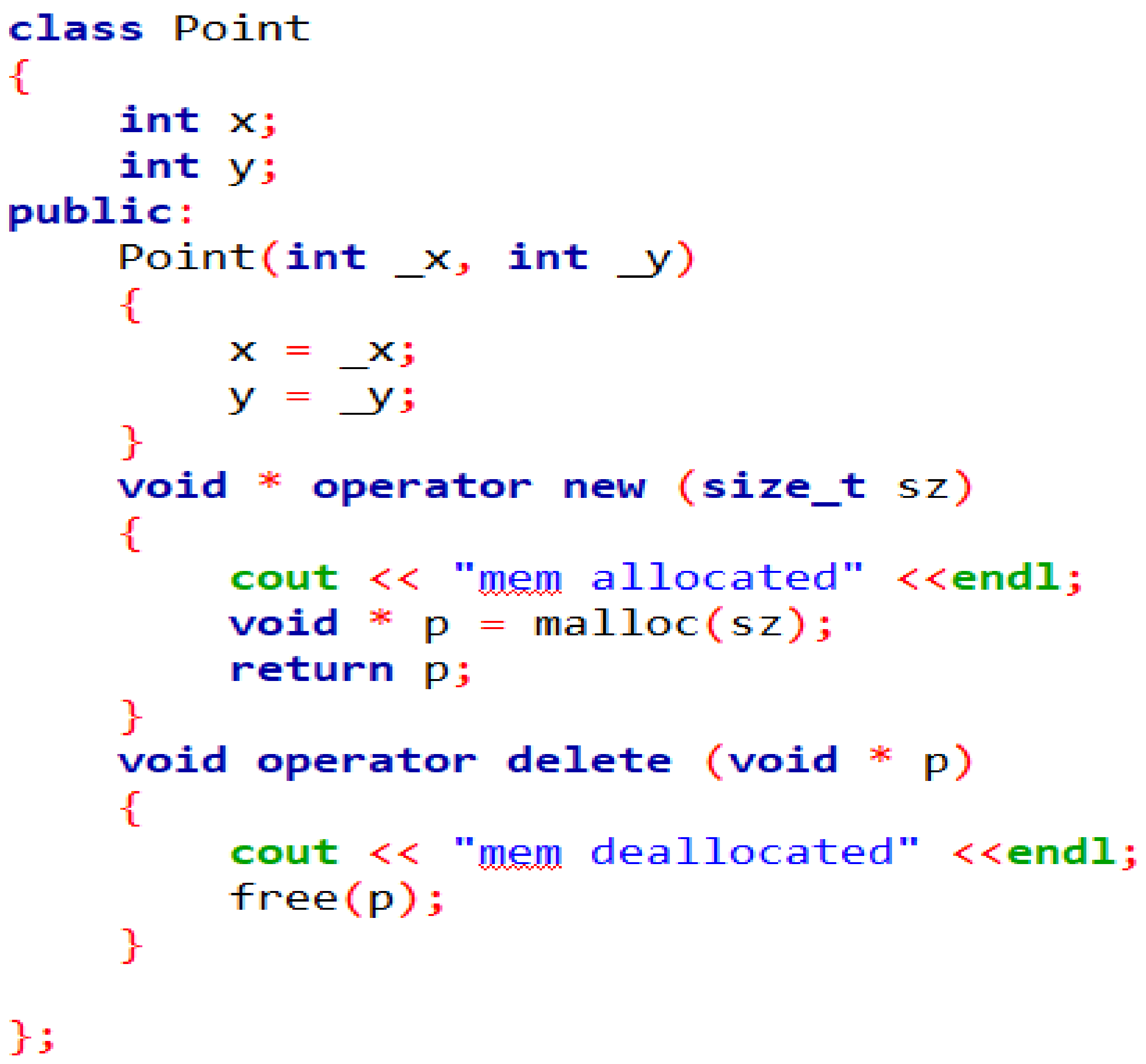
been performed.

- Returns a value of type *type*
- No parameter can be specified
- Conversion function must be a member function

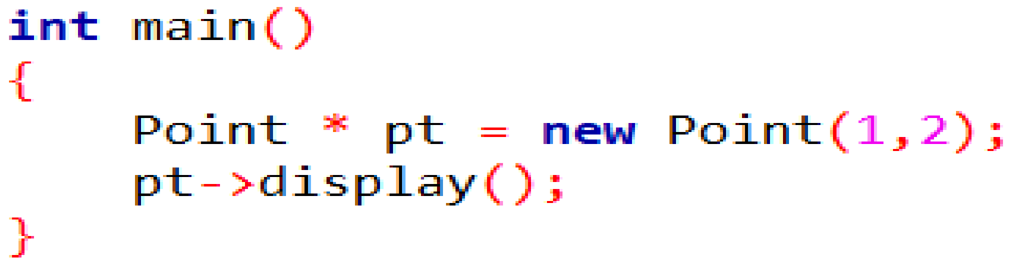
Overloading new and delete

```
void    *   operator    new     (size_t     count);
void    operator    delete      (void   *   ptr);
```

Overloading new and delete



```
class Point
{
    int x;
    int y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void * operator new (size_t sz)
    {
        cout << "mem allocated" <<endl;
        void * p = malloc(sz);
        return p;
    }
    void operator delete (void * p)
    {
        cout << "mem deallocated" <<endl;
        free(p);
    }

};
```

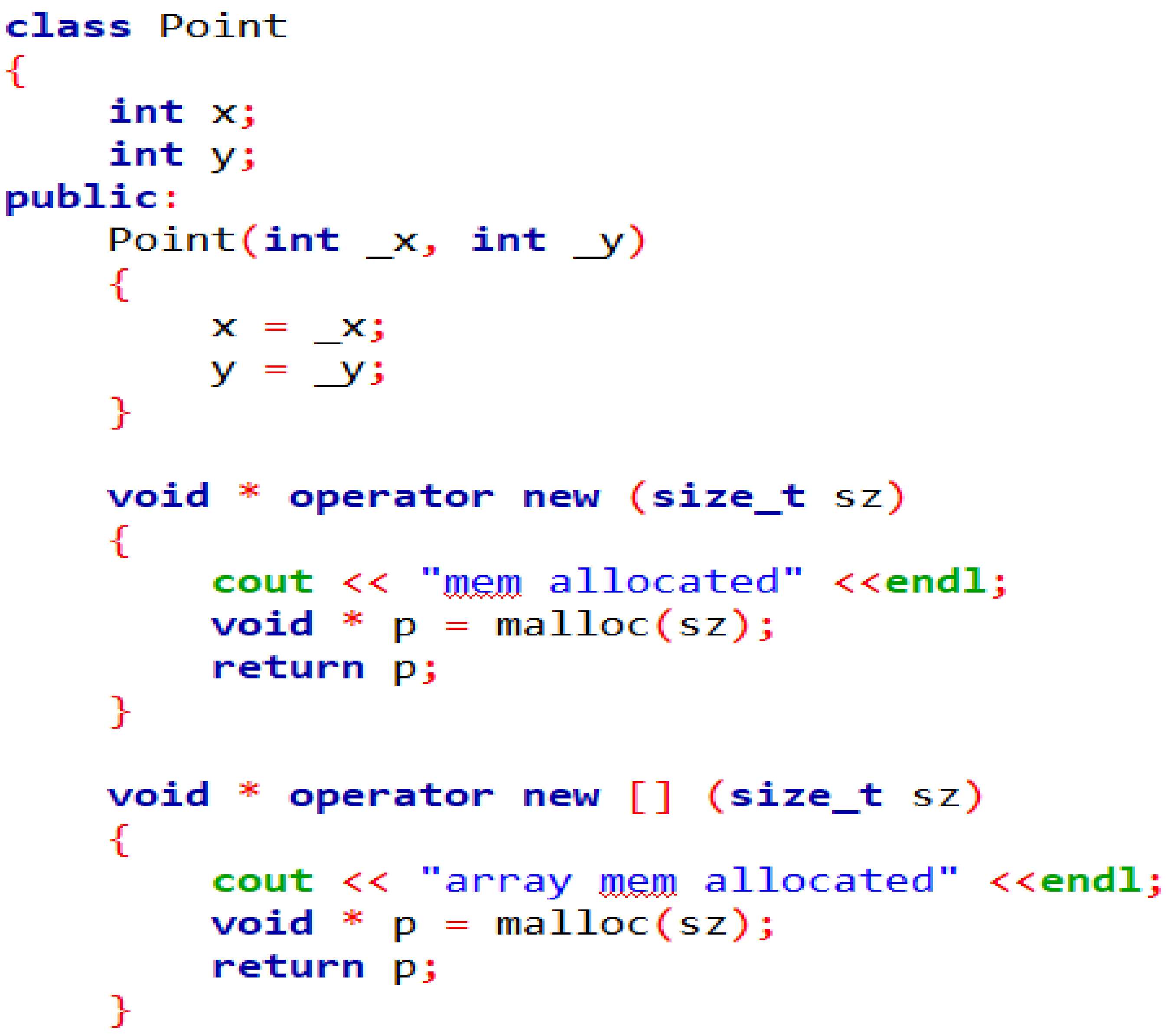


```
int main()
{
    Point * pt = new Point(1,2);
    pt->display();
}
```

Overloading new [] and delete []

```
void   *   operator   new   []  (size_t     count);
void   operator   delete    []      (void   *   ptr);
```

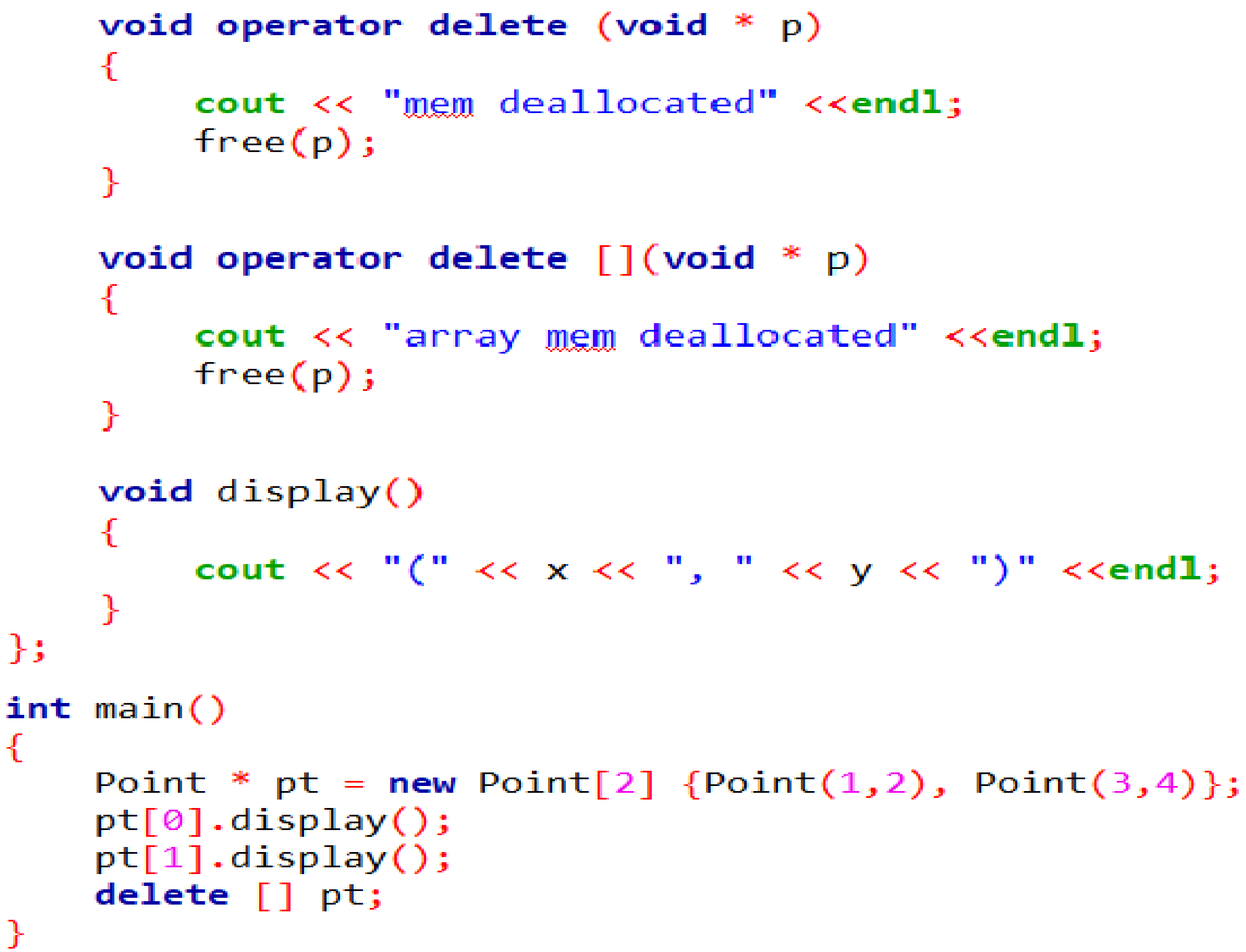
Overloading new [] and delete []



```
class Point
{
    int x;
    int y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }

    void * operator new (size_t sz)
    {
        cout << "mem allocated" <<endl;
        void * p = malloc(sz);
        return p;
    }

    void * operator new [] (size_t sz)
    {
        cout << "array mem allocated" <<endl;
        void * p = malloc(sz);
        return p;
    }
```

Overloading new [] and delete []

Cont...



```
    void operator delete (void * p)
    {
        cout << "mem deallocated" <<endl;
        free(p);
    }

    void operator delete [](void * p)
    {
        cout << "array mem deallocated" <<endl;
        free(p);
    }

    void display()
    {
        cout << "(" << x << ", " << y << ")" <<endl;
    }
};

int main()
{
    Point * pt = new Point[2] {Point(1,2), Point(3,4)};
    pt[0].display();
    pt[1].display();
    delete [] pt;
}
```

References

- www.cs.bu.edu/fac/gkollios/cs113/Slides/lecture12.ppt
- Teach Yourself C++, 3rd Ed. By Herb Schildt (Chapter 6)
- www.tutorialspoint.com/cplusplus/cpp_overloading.htm
- https://en.wikibooks.org/wiki/C%2B%2B_Programming/Operators/Operator_Overloading

Thank you