Creative Software Design

Copy Constructor, Operator Overloading

Yunho Kim
yunhokim@hanyang.ac.kr
Dept. of Computer Science

Today's Topics

- Copy constructor
- `friend`, `static`
- Operator overloading

Copy Constructor

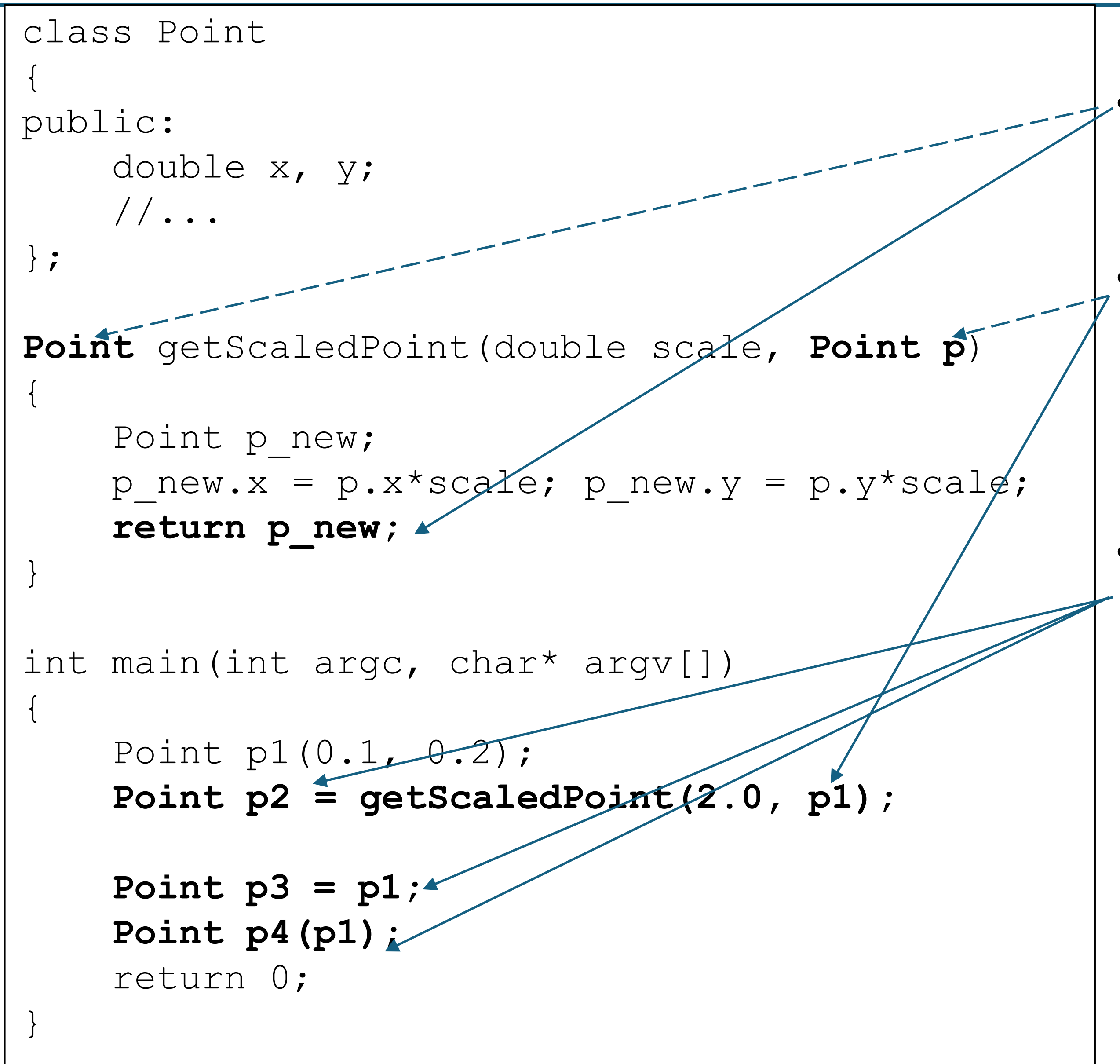
- A copy constructor is a constructor that initializes an object using another object of the same class.

```
ClassName(const ClassName& src_obj);
```

When is a Copy Constructor Called?

- When an object is returned by value. (before C++11)
  - Since C++11, a copy constructor may not be called due to Return Value Optimization (RVO)
- When an object is passed by value (not by address value) as a function argument.
- When an object is constructed based on another object of the same class.

When is a Copy Constructor Called?



```
class Point
{
public:
    double x, y;
    //...
};

Point getScaledPoint(double scale, Point p)
{
    Point p_new;
    p_new.x = p.x*scale; p_new.y = p.y*scale;
    return p_new;
}

int main(int argc, char* argv[])
{
    Point p1(0.1, 0.2);
    Point p2 = getScaledPoint(2.0, p1);

    Point p3 = p1;
    Point p4(p1);
    return 0;
}
```

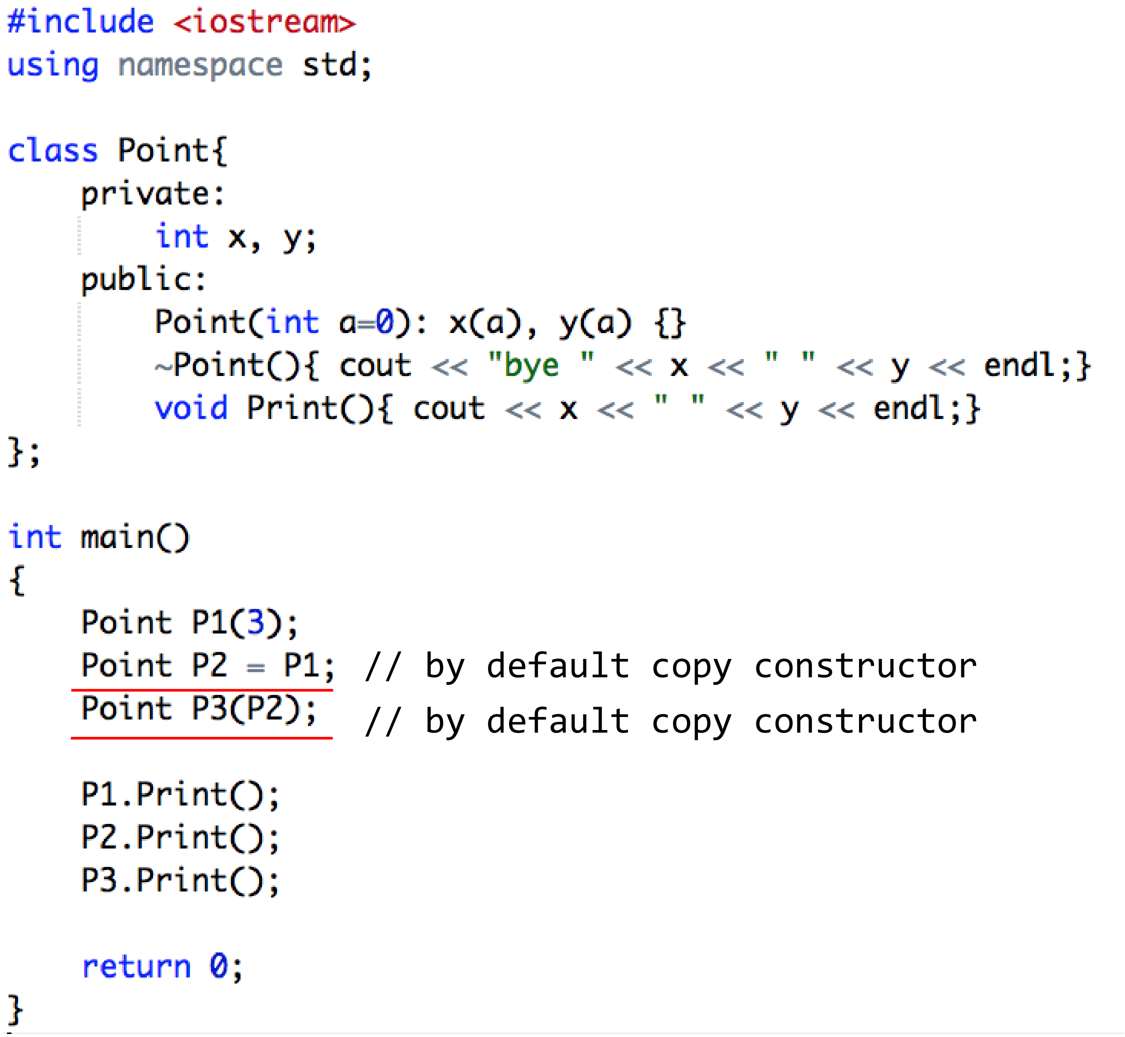
- When an object is returned by value.
- When an object is passed by value (not by address value) as a function argument.
- When an object is constructed based on another object of the same class.

Default Copy Constructor

- A default copy constructor is implicitly created by compiler if there is no user-defined copy constructor.

- It does a member-wise copy between objects,
  - where each member is copied by its own copy constructor.
  - This works fine in general, but does not work for some cases. We should define our own copy constructor for these cases.

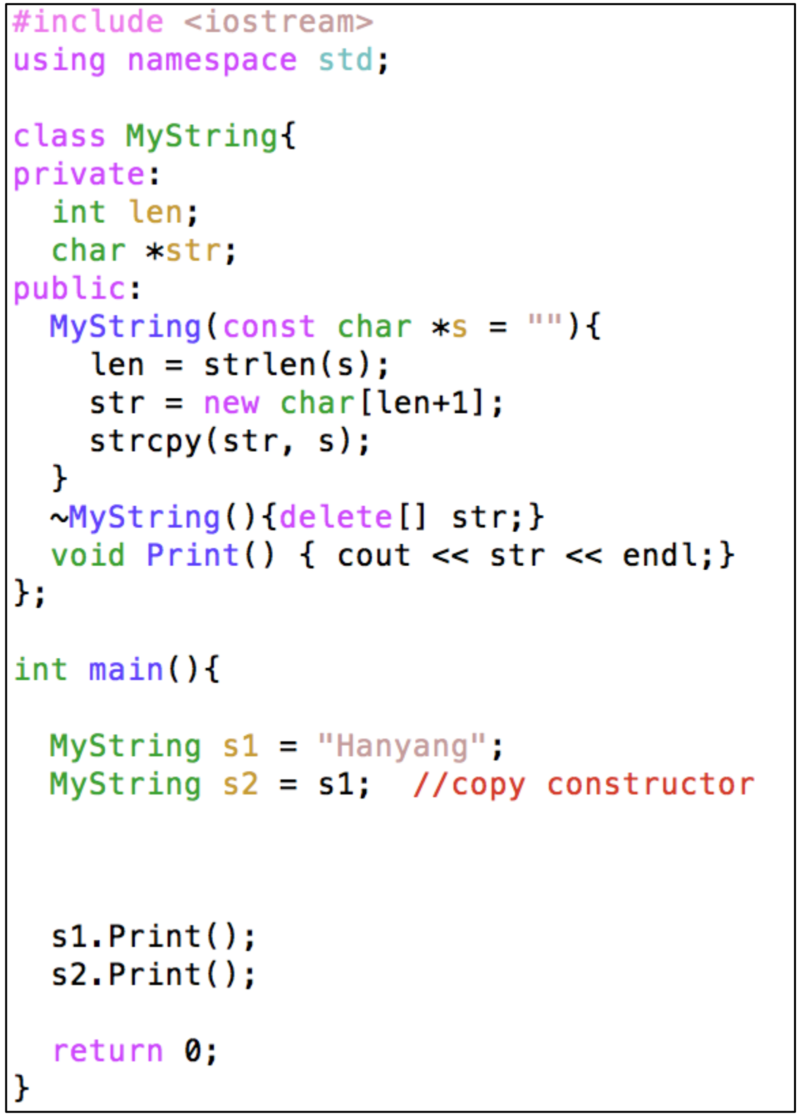
Default Copy Constructor: Example 1



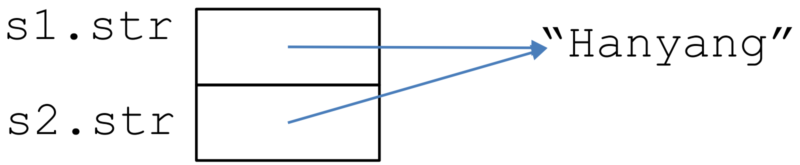
```cpp
#include <iostream>
using namespace std;

class Point{
    private:
        int x, y;
    public:
        Point(int a=0): x(a), y(a) {}
        ~Point(){ cout << "bye " << x << " " << y << endl;}
        void Print(){ cout << x << " " << y << endl;}
};

int main()
{
    Point P1(3);
    Point P2 = P1; // by default copy constructor
    Point P3(P2);  // by default copy constructor

    P1.Print();
    P2.Print();
    P3.Print();

    return 0;
}
```

- Default copy constructor copies each member of the object

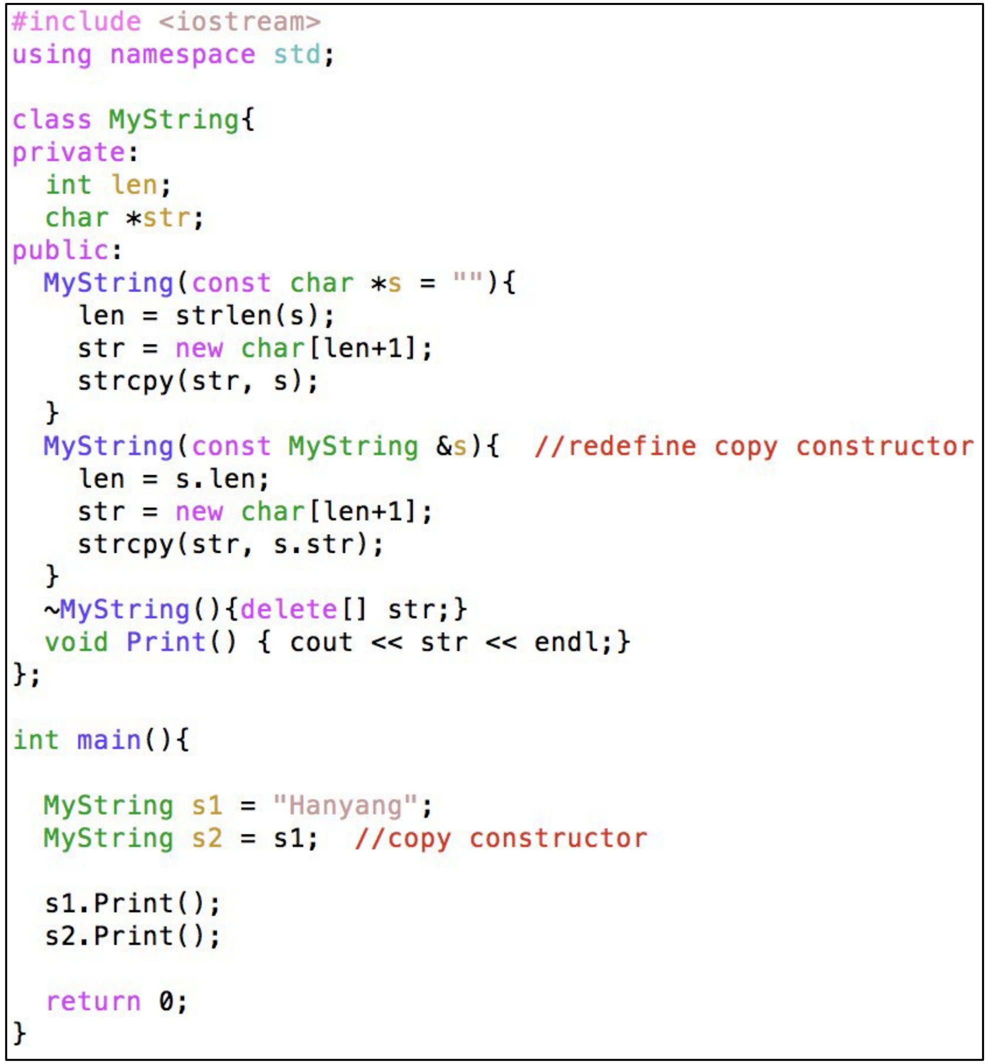
Default Copy Constructor: Example 2-1



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

class MyString{
private:
  int len;
  char *str;
public:
  MyString(const char *s = ""){
    len = strlen(s);
    str = new char[len+1];
    strcpy(str, s);
  }
  ~MyString(){delete[] str;}
  void Print() { cout << str << endl;}
};

int main(){

  MyString s1 = "Hanyang";
  MyString s2 = s1;   //copy constructor



  s1.Print();
  s2.Print();

  return 0;
}
```

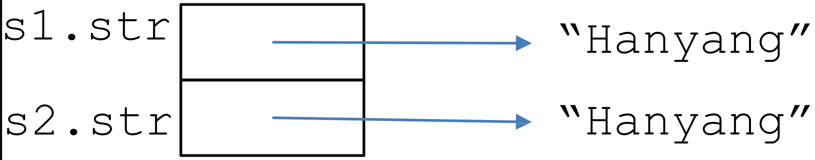


User-defined Copy Constructor: Example 2-2

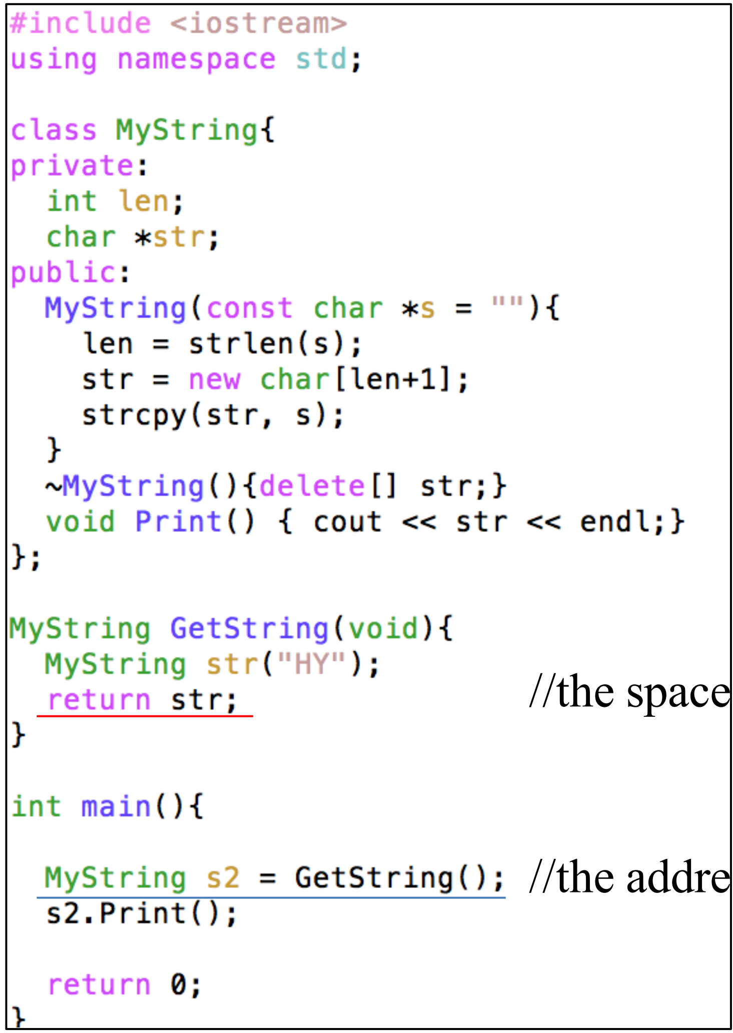


```cpp
#include <iostream>
using namespace std;

class MyString{
private:
  int len;
  char *str;
public:
  MyString(const char *s = ""){
    len = strlen(s);
    str = new char[len+1];
    strcpy(str, s);
  }
  MyString(const MyString &s){  //redefine copy constructor
    len = s.len;
    str = new char[len+1];
    strcpy(str, s.str);
  }
  ~MyString(){delete[] str;}
  void Print() { cout << str << endl;}
};

int main(){

  MyString s1 = "Hanyang";
  MyString s2 = s1;  //copy constructor

  s1.Print();
  s2.Print();

  return 0;
}
```

- The problem of deallocation by delete operator was resolved



Default Copy Constructor: Example 3



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

class MyString{
private:
  int len;
  char *str;
public:
  MyString(const char *s = ""){
    len = strlen(s);
    str = new char[len+1];
    strcpy(str, s);
  }
  ~MyString(){delete[] str;}
  void Print() { cout << str << endl;}
};

MyString GetString(void){
  MyString str("HY");
  return str;
}

int main(){

  MyString s2 = GetString();
  s2.Print();

  return 0;
}
```

How many times MyString::~MyString() is invoked?

//the space for ”HY” is deallocated

//the address to ”HY” is copied

Default Copy Constructor & Default Constructor

- Recall: A default constructor is implicitly created by compiler if there is no user-defined constructor.

- If you define a copy constructor, the complier doesn’t create the default constructor and default copy constructor.

Copy Constructor: Example

```
class Point
{
public:
    double x, y;
    Point(double x_, double y_):x(x_), y(y_) {}

    // The most popular form.
    Point(const Point& p) { x = p.x; y = p.y; }

    // You can also use this form.
    // But copy constructor generally doesn't need to update
    // the passed object, so the first form is the most popular.
    Point(Point& p) { x = p.x; y = p.y; }

    // Compile error. If it were compiled, it would result in
    // infinite calling of copy constructor.
    Point(Point p) { x = p.x; y = p.y; }
};
```

Friend Class and Function

- Functions or classes can be "friends" of another class (let's say ClassA).
  - If you declare them as "friends" in the definition of ClassA,
  - Then these "friends" can access all members of ClassA including private members.

```
class ClassA {
 private:
  int var_;
  friend class ClassB;
  friend void DoSomething(const ClassA& a);
};

class ClassB {
  // ...
  void Function(const ClassA& a) { cout << a.var_; }  // OK.
};

void DoSomething(const ClassA& a) { cout << a.var_; }  // OK.
```

Friend Class and Function

- Note that access specifiers have no effect on the meaning of friend declarations
  - They can appear in private, protected, or public sections, with no difference
- – https://en.cppreference.com/w/cpp/language/friend

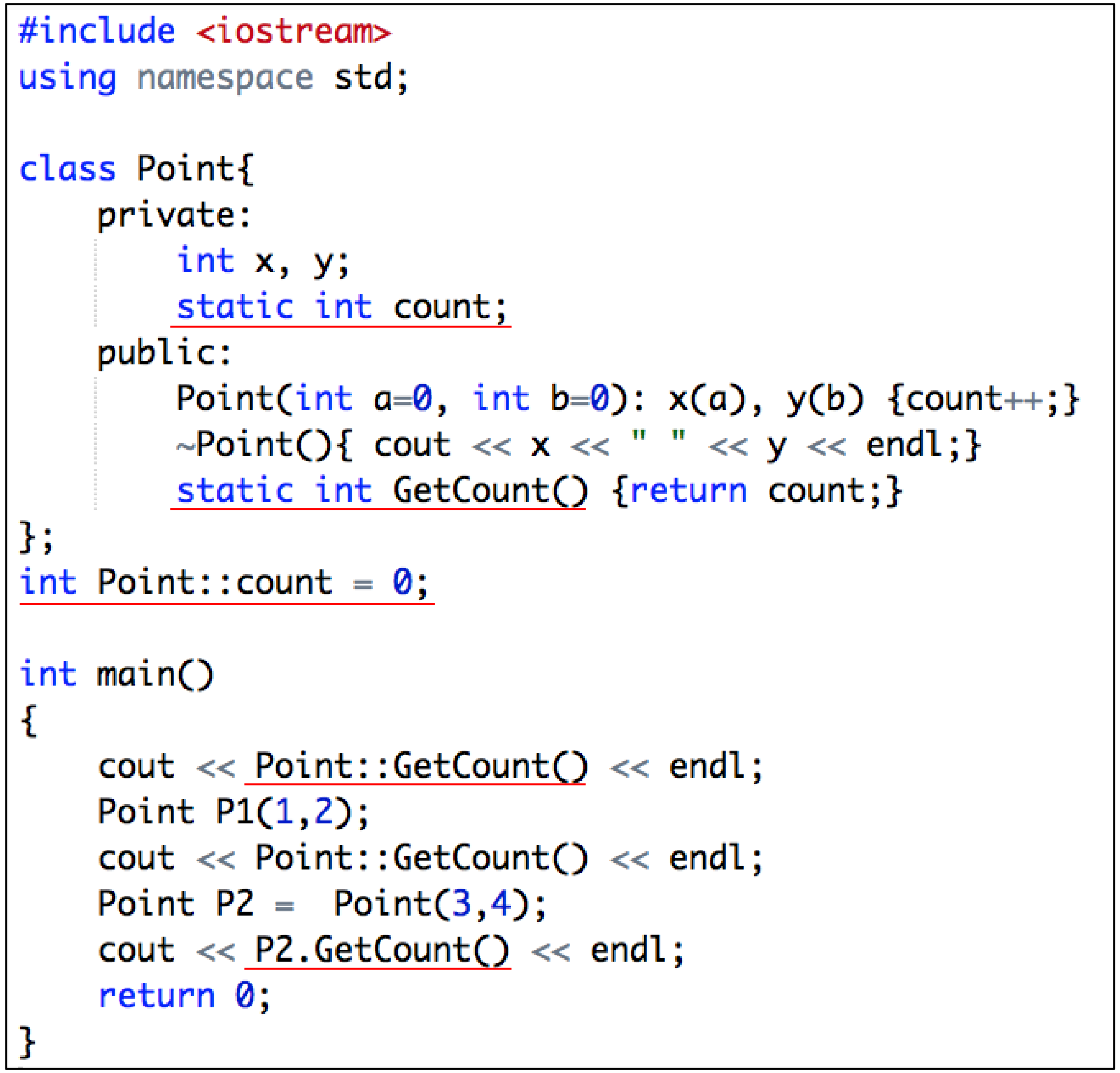
Static Members

- Static members (variables and functions) in a class are **shared by all the objects of the class.**
  - Static member functions can only access static members.
  - Static member functions cannot be virtual.

- Static members can be accessed by class name or object name.

- Static member variables are defined outside the class.

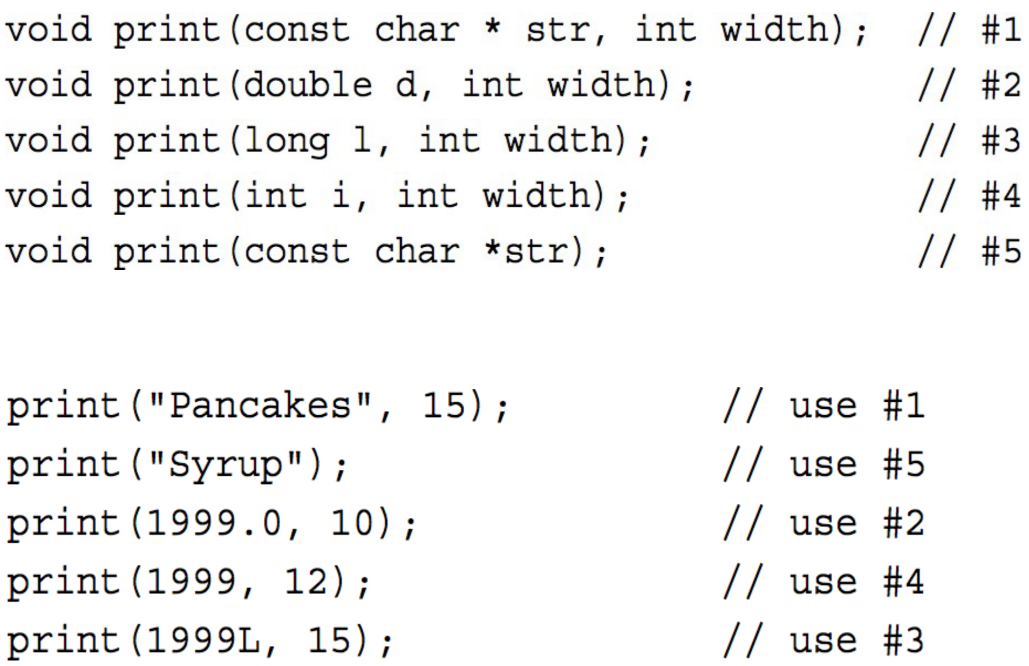
Static Members



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

class Point{
    private:
        int x, y;
        static int count;
    public:
        Point(int a=0, int b=0): x(a), y(b) {count++;}
        ~Point(){ cout << x << " " << y << endl;}
        static int GetCount() {return count;}
};
int Point::count = 0;

int main()
{
    cout << Point::GetCount() << endl;
    Point P1(1,2);
    cout << Point::GetCount() << endl;
    Point P2 =  Point(3,4);
    cout << P2.GetCount() << endl;
    return 0;
}
```

Recall: Function Overloading

- Use multiple functions sharing the same name
  - A family of functions that do the same thing but using different argument lists



```
void print(const char * str, int width);  // #1
void print(double d, int width);          // #2
void print(long l, int width);            // #3
void print(int i, int width);             // #4
void print(const char *str);              // #5


print("Pancakes", 15);           // use #1
print("Syrup");                  // use #5
print(1999.0, 10);               // use #2
print(1999, 12);                 // use #4
print(1999L, 15);                // use #3
```

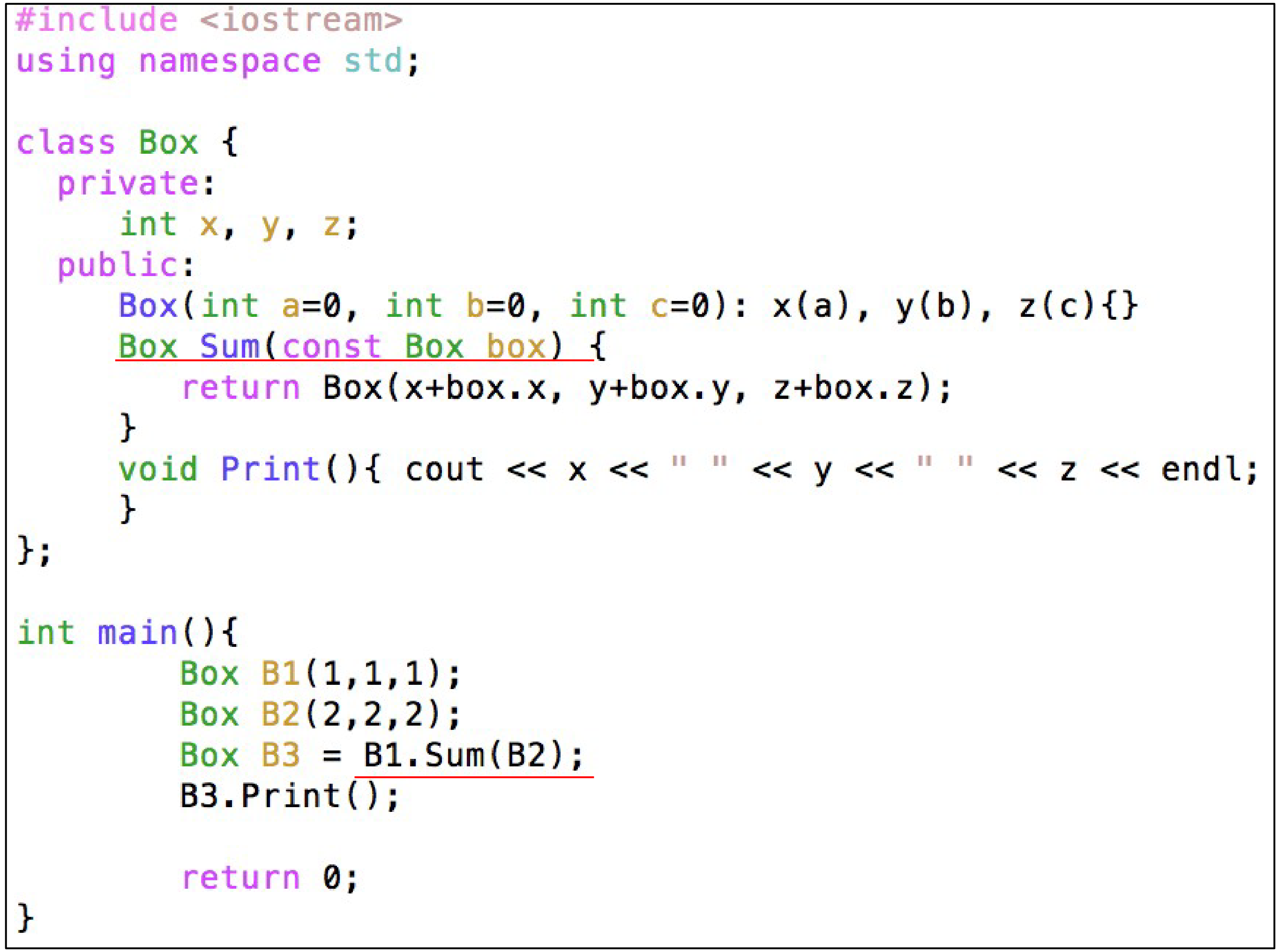
Operator Overloading

- An operator function is a special function form to overload an operator operator*op* (arguments)
  - *op* is a valid C++ operator
  - operator+() overloads the + operator

- Note that C++ even allows re-defining built-in operators such as +, -, *, ...

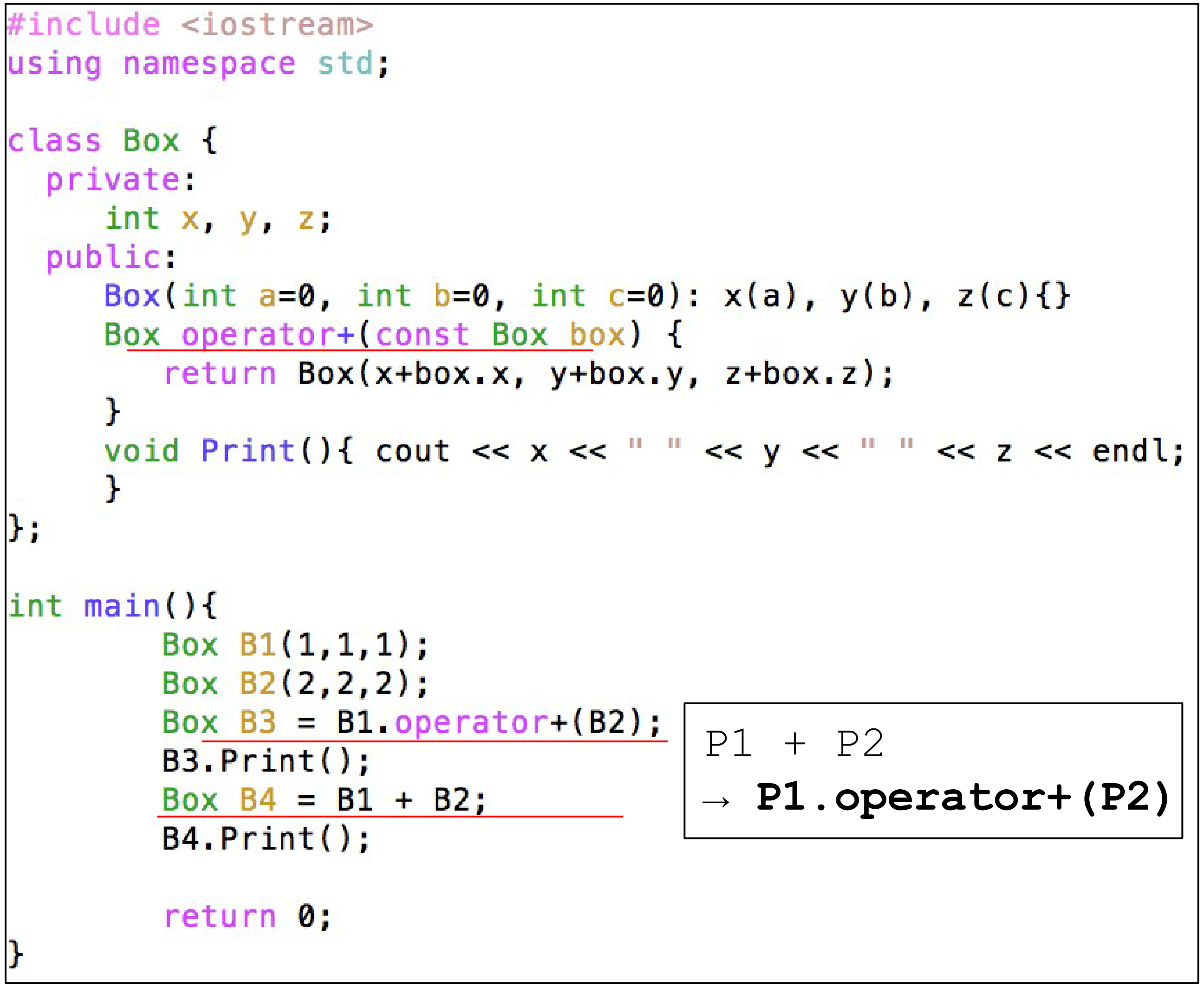
- An operator can be overloaded as a **class member function** or **non-member function**.

Operator Overloading as a Member Function



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

class Box {
  private:
     int x, y, z;
  public:
     Box(int a=0, int b=0, int c=0): x(a), y(b), z(c){}
     Box Sum(const Box box) {
        return Box(x+box.x, y+box.y, z+box.z);
     }
     void Print(){ cout << x << " " << y << " " << z << endl;
     }
};

int main(){
        Box B1(1,1,1);
        Box B2(2,2,2);
        Box B3 = B1.Sum(B2);
        B3.Print();

        return 0;
}
```

Operator Overloading as a Member Function



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

class Box {
  private:
     int x, y, z;
  public:
     Box(int a=0, int b=0, int c=0): x(a), y(b), z(c){}
     Box operator+(const Box box) {
        return Box(x+box.x, y+box.y, z+box.z);
     }
     void Print(){ cout << x << " " << y << " " << z << endl;
     }
};

int main(){
        Box B1(1,1,1);
        Box B2(2,2,2);
        Box B3 = B1.operator+(B2);
        B3.Print();
        Box B4 = B1 + B2;
        B4.Print();

        return 0;
}
```

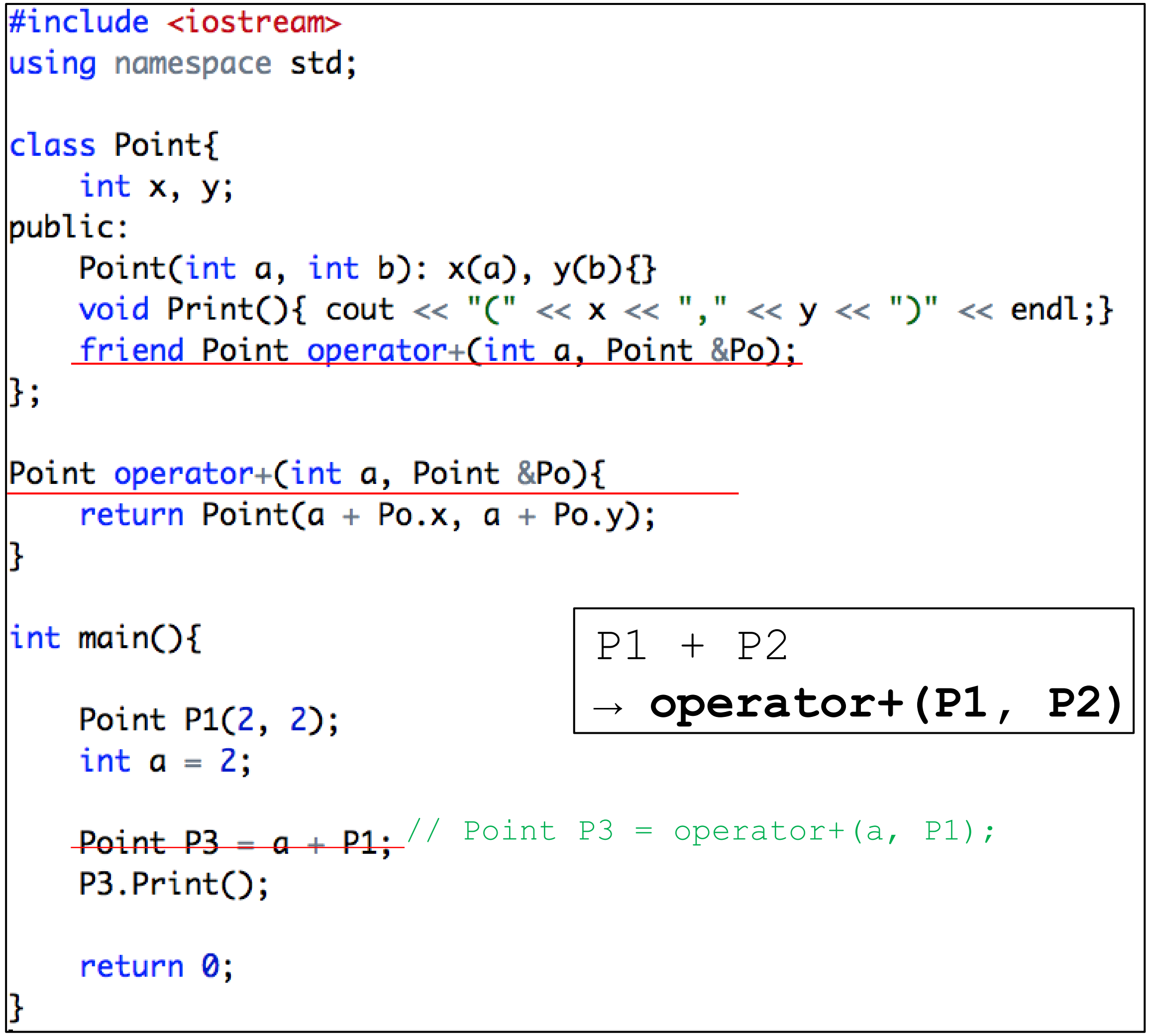
```
P1 + P2
→ P1.operator+(P2)
```

Operator Overloading as a Member Function

- `P1 + P2`→ **`P1.operator+(P2)`**
- That means, the operator overloaded member function gets invoked on the **first operand.**

- What if the **first operand is not a class type**, like `double`?
  - – For example, `2.0 + P2` ?
- →Use **non-member** operator overloaded function!

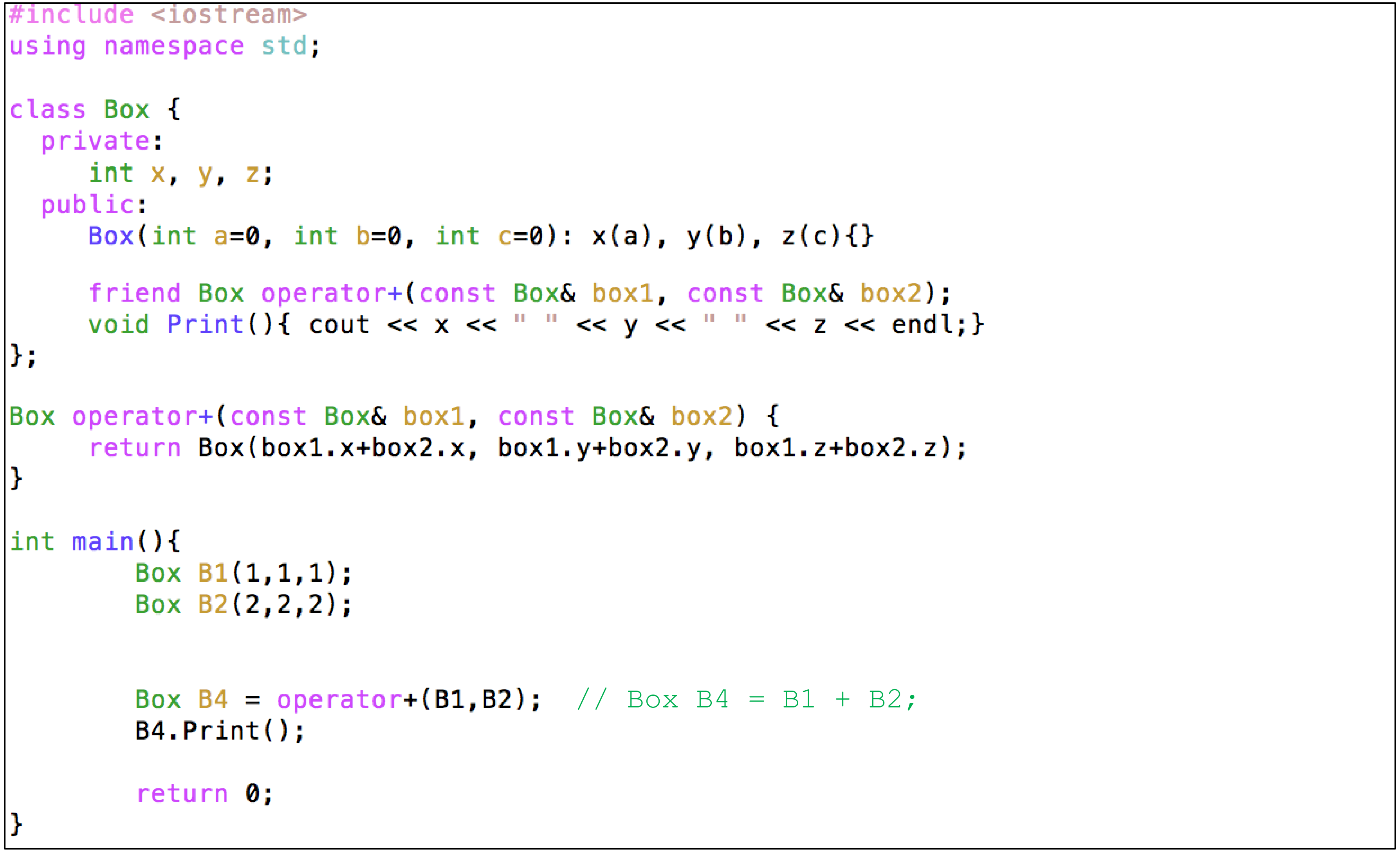
Operator Overloading as a Non-member Function



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

class Point{
    int x, y;
public:
    Point(int a, int b): x(a), y(b){}
    void Print(){ cout << "(" << x << "," << y << ")" << endl;}
    friend Point operator+(int a, Point &Po);
};

Point operator+(int a, Point &Po){
    return Point(a + Po.x, a + Po.y);
}

int main(){

    Point P1(2, 2);
    int a = 2;

    ~~Point P3 = a + P1;~~ // Point P3 = operator+(a, P1);
    P3.Print();

    return 0;
}
```

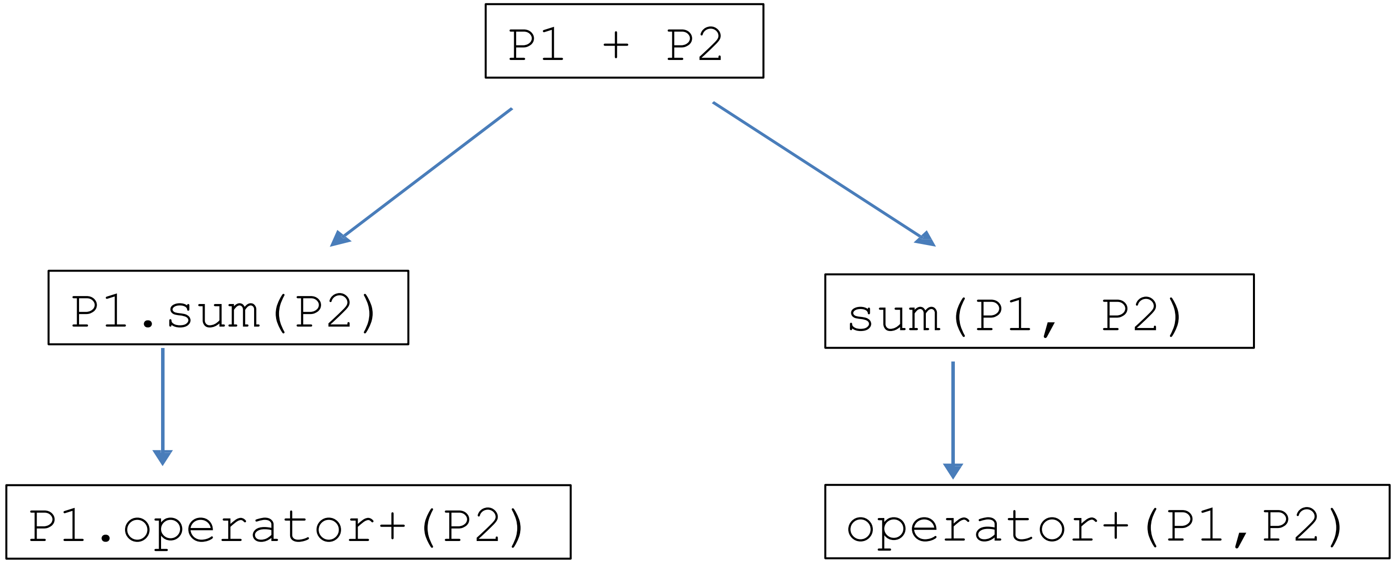
P1 + P2
→ **operator+(P1, P2)**

Operator Overloading as a Non-member Function



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

class Box {
  private:
     int x, y, z;
  public:
     Box(int a=0, int b=0, int c=0): x(a), y(b), z(c){}

     friend Box operator+(const Box& box1, const Box& box2);
     void Print(){ cout << x << " " << y << " " << z << endl;}
};

Box operator+(const Box& box1, const Box& box2) {
     return Box(box1.x+box2.x, box1.y+box2.y, box1.z+box2.z);
}

int main(){
        Box B1(1,1,1);
        Box B2(2,2,2);


        Box B4 = operator+(B1,B2);  // Box B4 = B1 + B2;
        B4.Print();

        return 0;
}
```

Operator Function



P1 + P2

P1.sum(P2)

sum(P1, P2)

P1.operator+(P2)

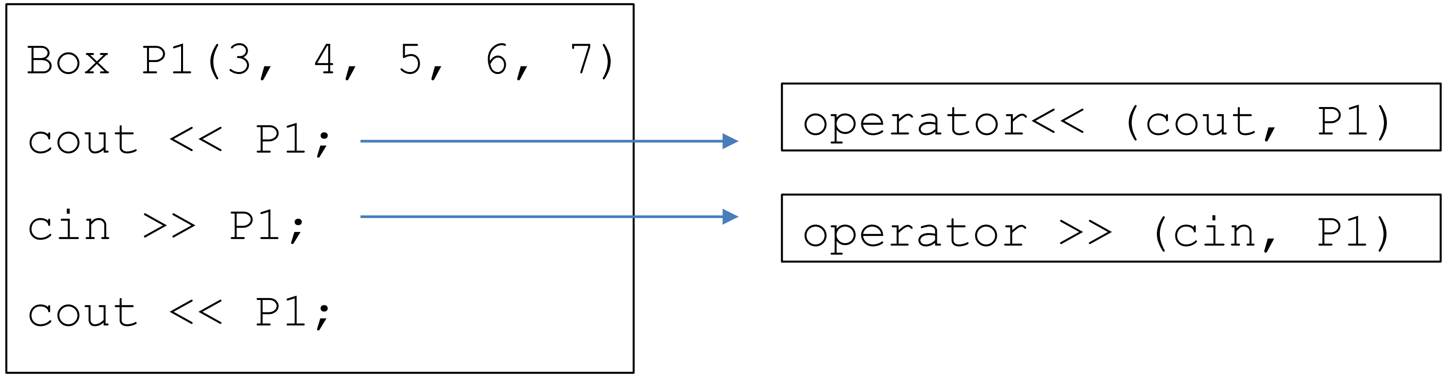
operator+(P1,P2)

as a member function

as a non-member function

Operator Overloading: <<, >> operator

- As a non-member function

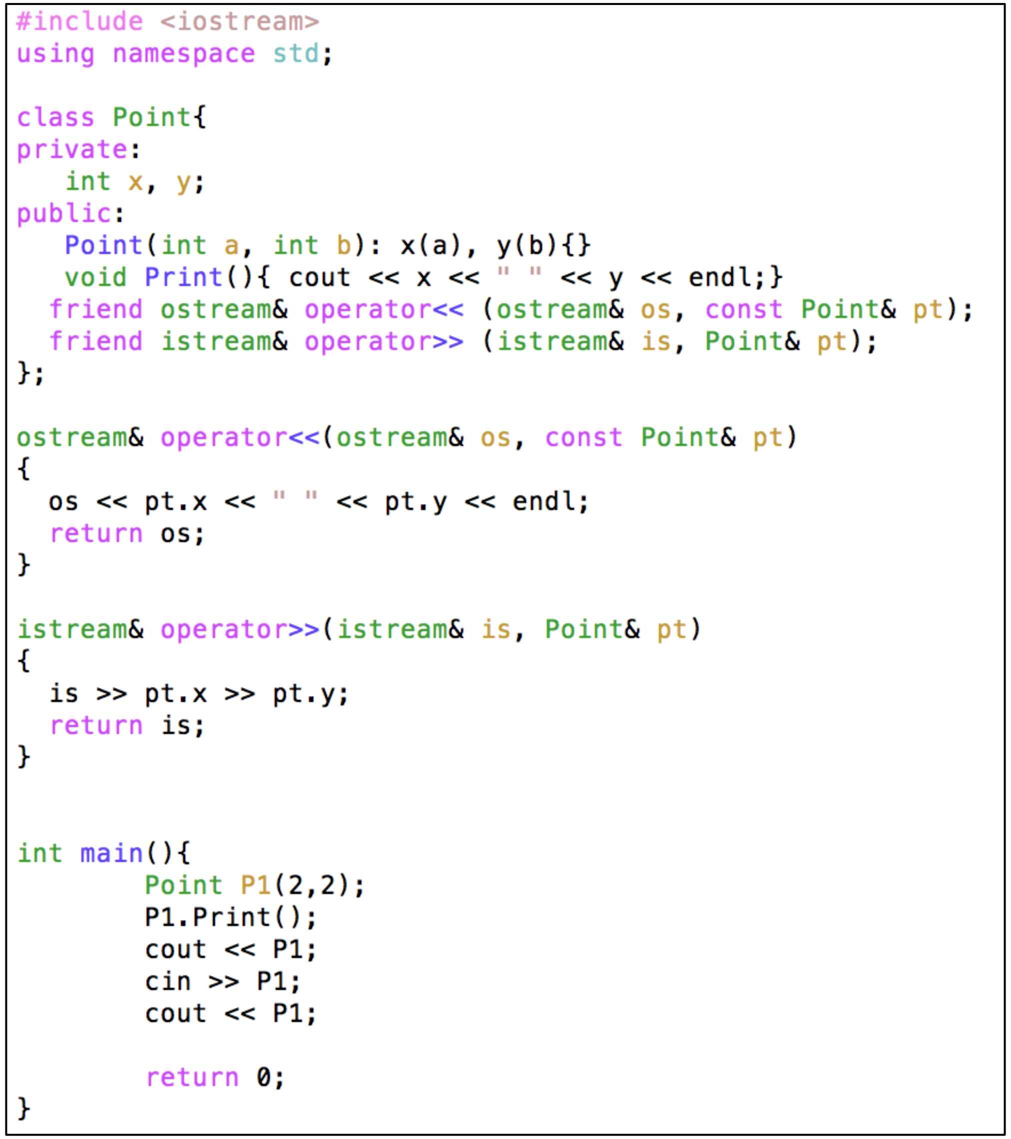


```
Box P1(3, 4, 5, 6, 7)
cout << P1;
cin >> P1;
cout << P1;
```

```
operator<< (cout, P1)
```

```
operator >> (cin, P1)
```

Operator Overloading: <<, >> operator



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

class Point{
private:
   int x, y;
public:
   Point(int a, int b): x(a), y(b){}
   void Print(){ cout << x << " " << y << endl;}
  friend ostream& operator<< (ostream& os, const Point& pt);
  friend istream& operator>> (istream& is, Point& pt);
};

ostream& operator<<(ostream& os, const Point& pt)
{
  os << pt.x << " " << pt.y << endl;
  return os;
}

istream& operator>>(istream& is, Point& pt)
{
  is >> pt.x >> pt.y;
  return is;
}


int main(){
        Point P1(2,2);
        P1.Print();
        cout << P1;
        cin >> P1;
        cout << P1;

        return 0;
}
```

Assignment Operator(= operator) Overloading

- A default assignment operator is implicitly created by compiler if there is no user-defined assignment operator.

- It does a member-wise copy between objects.
  - where each member is copied by its own assignment operator.
  - Like default copy constructor, this works fine in general, but does not work for some cases.

Copy Constructor vs. Assignment Operator

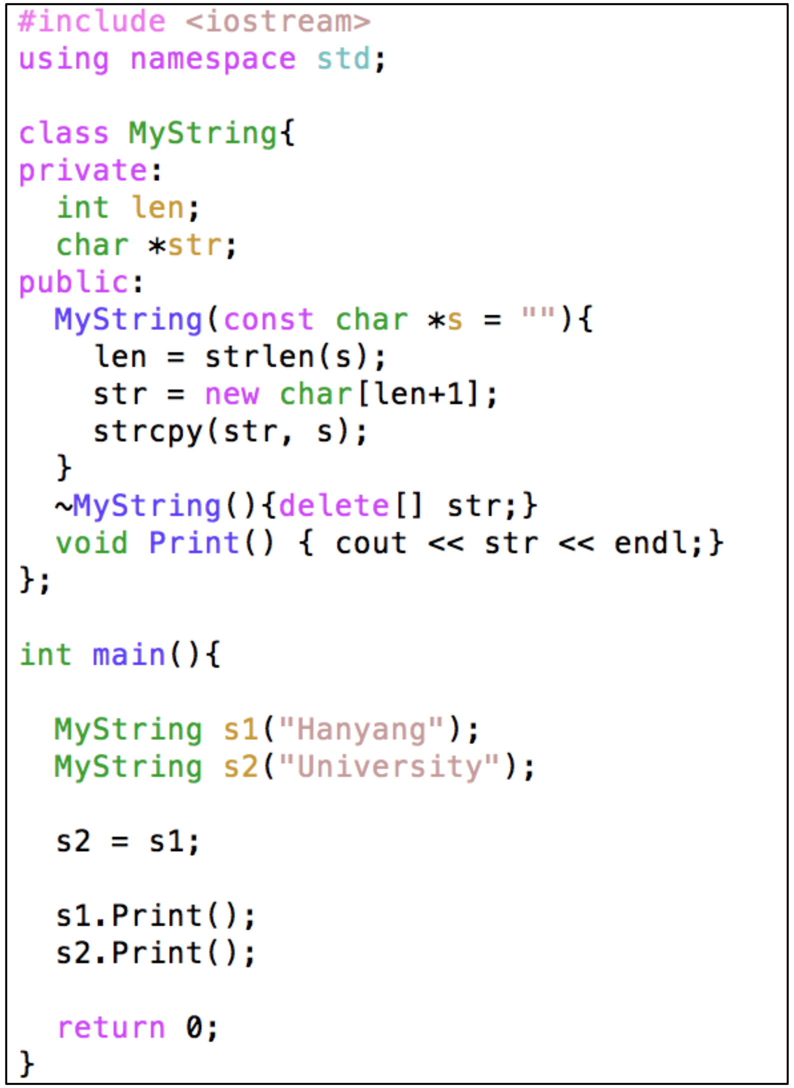
```
#include <iostream>
using namespace std;

class Point{
private:
    double x, y;
public:
    Point(double x_, double y_):x(x_), y(y_) {}

    Point(const Point& p)
    { x = p.x; y = p.y; cout << "copy constructor" << endl; }

    Point& operator=(const Point& p)
    { x = p.x; y = p.y; cout << "assignment operator" << endl; return *this; }
};
int main()
{
    Point p1(1,2);
    Point p2(p1);   // "copy constructor"
    Point p3 = p1;  // "copy constructor"

    Point p4(2,3);
    p4 = p1;        // "assignment operator"

    return 0;
}
```

Return Type of Assignment Operator

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

class Point
{
private:
  double x, y;
public:
  Point():x(0.0), y(0.0) {}
  Point(double x_, double y_):x(x_), y(y_) {}

  // inconsistent behavior with default assignment
  // operator and assignments for primitive types
  Point operator=(const Point& p){
    x = p.x; y = p.y; return Point(*this); }

  // same behavior as default assignment operator
  // and assignments for primitive types->
  // Use this!
  Point& operator=(const Point& p){
    x = p.x; y = p.y; return *this; }

friend ostream& operator<< (ostream& os,
                            const Point& p);
};
```

```cpp
ostream& operator<<(ostream& os, const Point& p)
{
  os << "(" << p.x << ", " << p.y << ")";
  return os;
}

int main()
{
  Point p1(1,2);
  Point p2, p3;
  (p3 = p2) = p1;

  cout << p1 << p2 << p3 << endl;
  return 0;
}
```

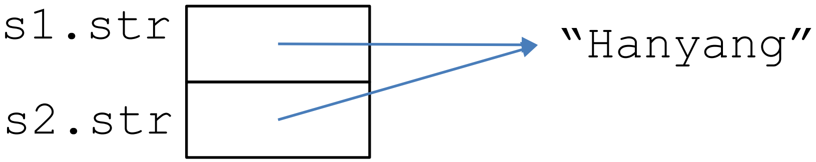
Default Assignment Operator: Example



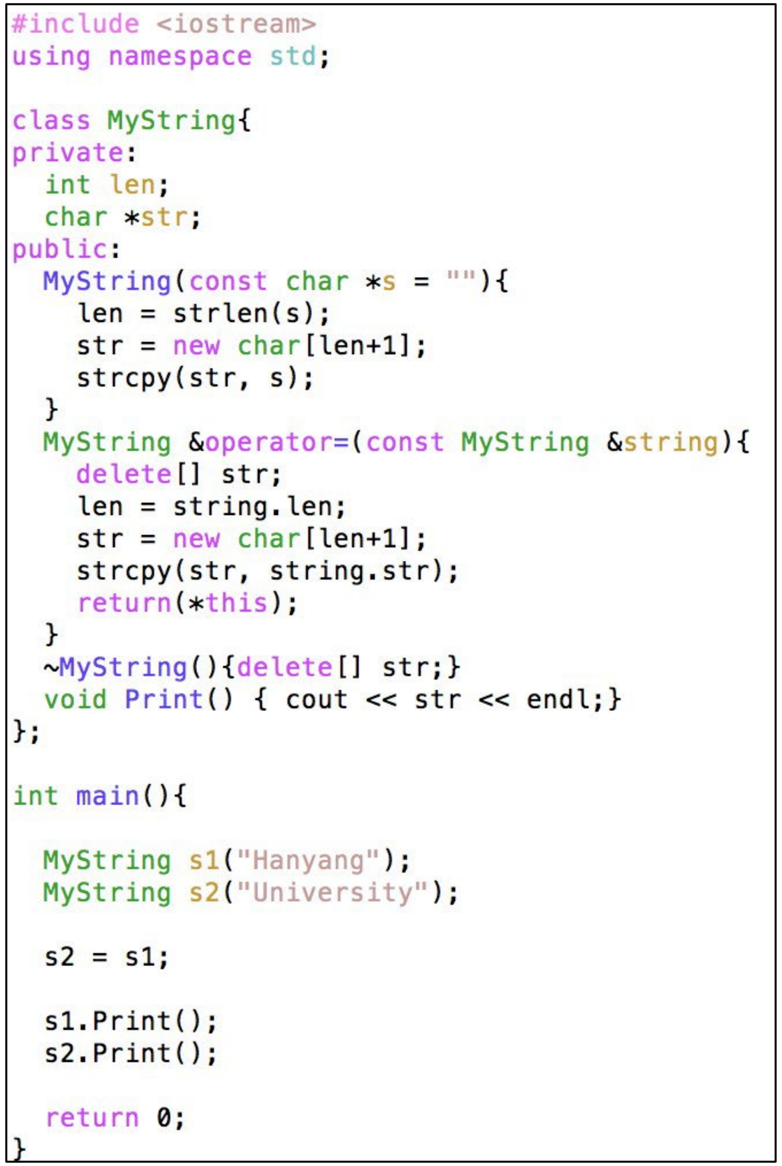
```
#include <iostream>
using namespace std;

class MyString{
private:
  int len;
  char *str;
public:
  MyString(const char *s = ""){
    len = strlen(s);
    str = new char[len+1];
    strcpy(str, s);
  }
  ~MyString(){delete[] str;}
  void Print() { cout << str << endl;}
};

int main(){

  MyString s1("Hanyang");
  MyString s2("University");

  s2 = s1;

  s1.Print();
  s2.Print();

  return 0;
}
```

Is it OK?

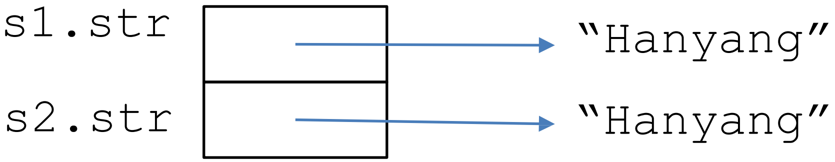
Operator= copies the address



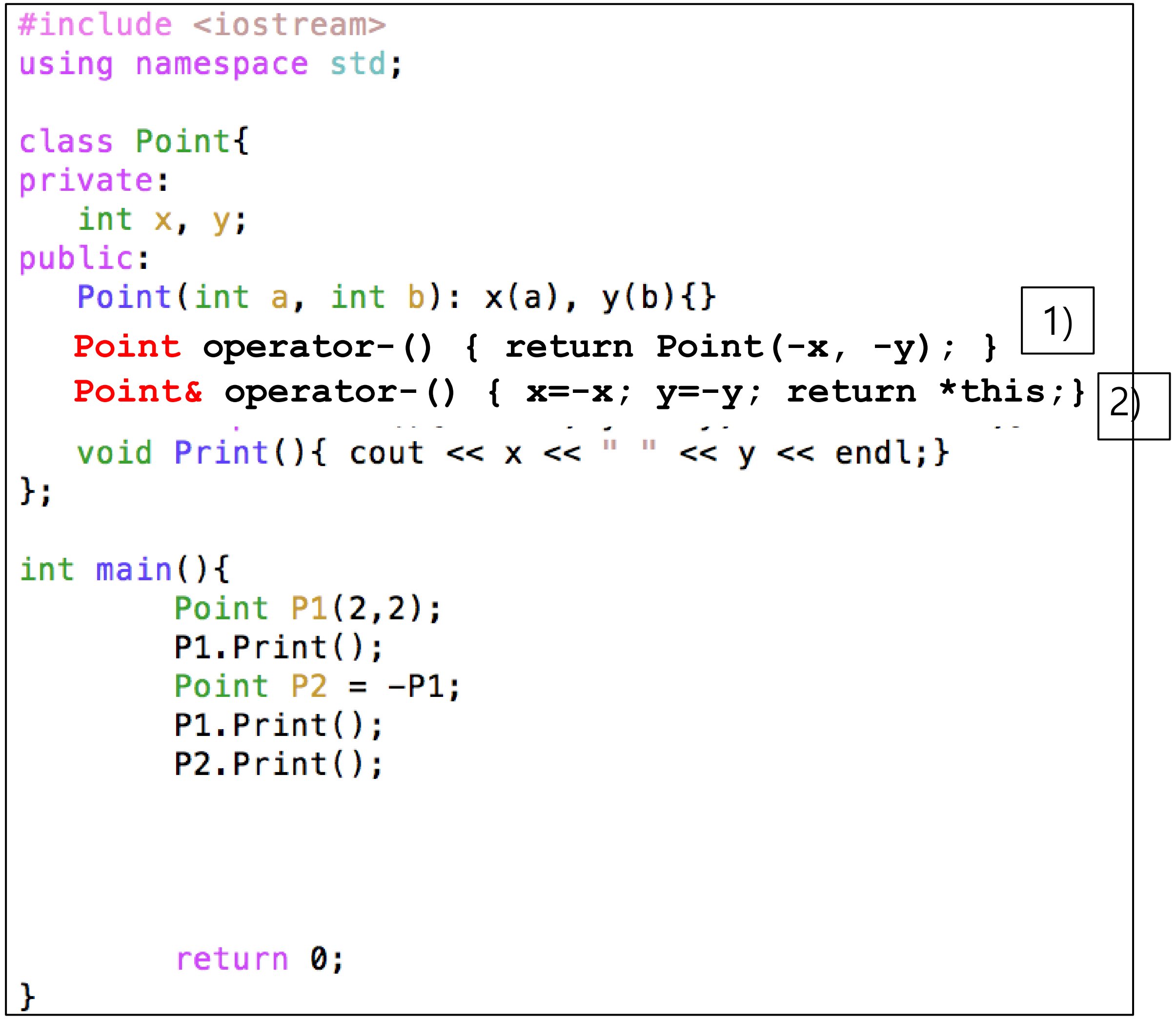
User-defined Assignment Operator: Example



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

class MyString{
private:
  int len;
  char *str;
public:
  MyString(const char *s = ""){
    len = strlen(s);
    str = new char[len+1];
    strcpy(str, s);
  }
  MyString &operator=(const MyString &string){
    delete[] str;
    len = string.len;
    str = new char[len+1];
    strcpy(str, string.str);
    return(*this);
  }
  ~MyString(){delete[] str;}
  void Print() { cout << str << endl;}
};

int main(){

  MyString s1("Hanyang");
  MyString s2("University");

  s2 = s1;

  s1.Print();
  s2.Print();

  return 0;
}
```



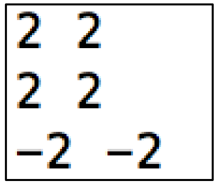
Operator Overloading: Unary Operator



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

class Point{
private:
   int x, y;
public:
   Point(int a, int b): x(a), y(b){}
   Point operator-() { return Point(-x, -y); }   1)
   Point& operator-() { x=-x; y=-y; return *this;}   2)
   void Print(){ cout << x << " " << y << endl;}
};

int main(){
        Point P1(2,2);
        P1.Print();
        Point P2 = -P1;
        P1.Print();
        P2.Print();




        return 0;
}
```

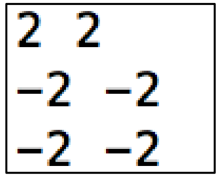
1) is consistent with primitive types.

1)



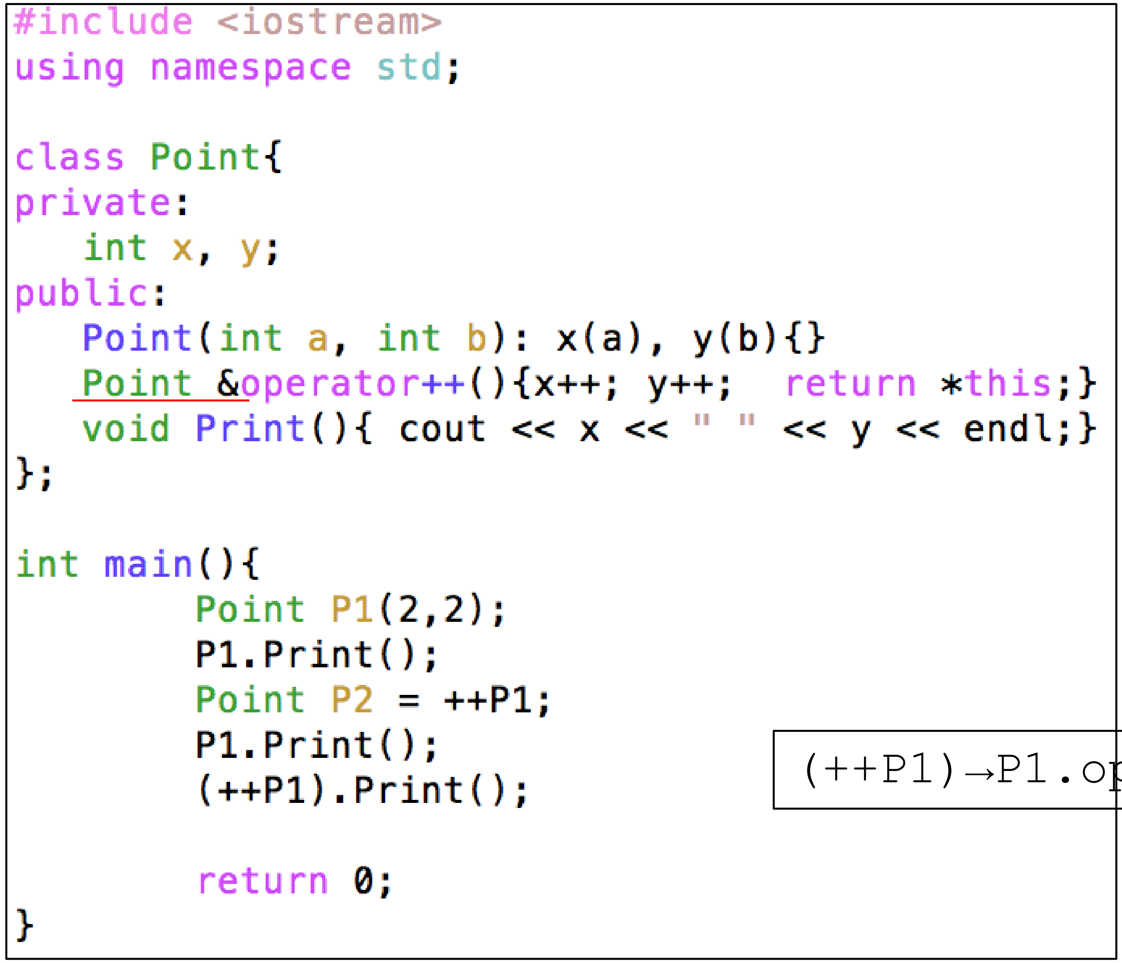
```
2 2
2 2
-2 -2
```

2)



```
2 2
-2 -2
-2 -2
```

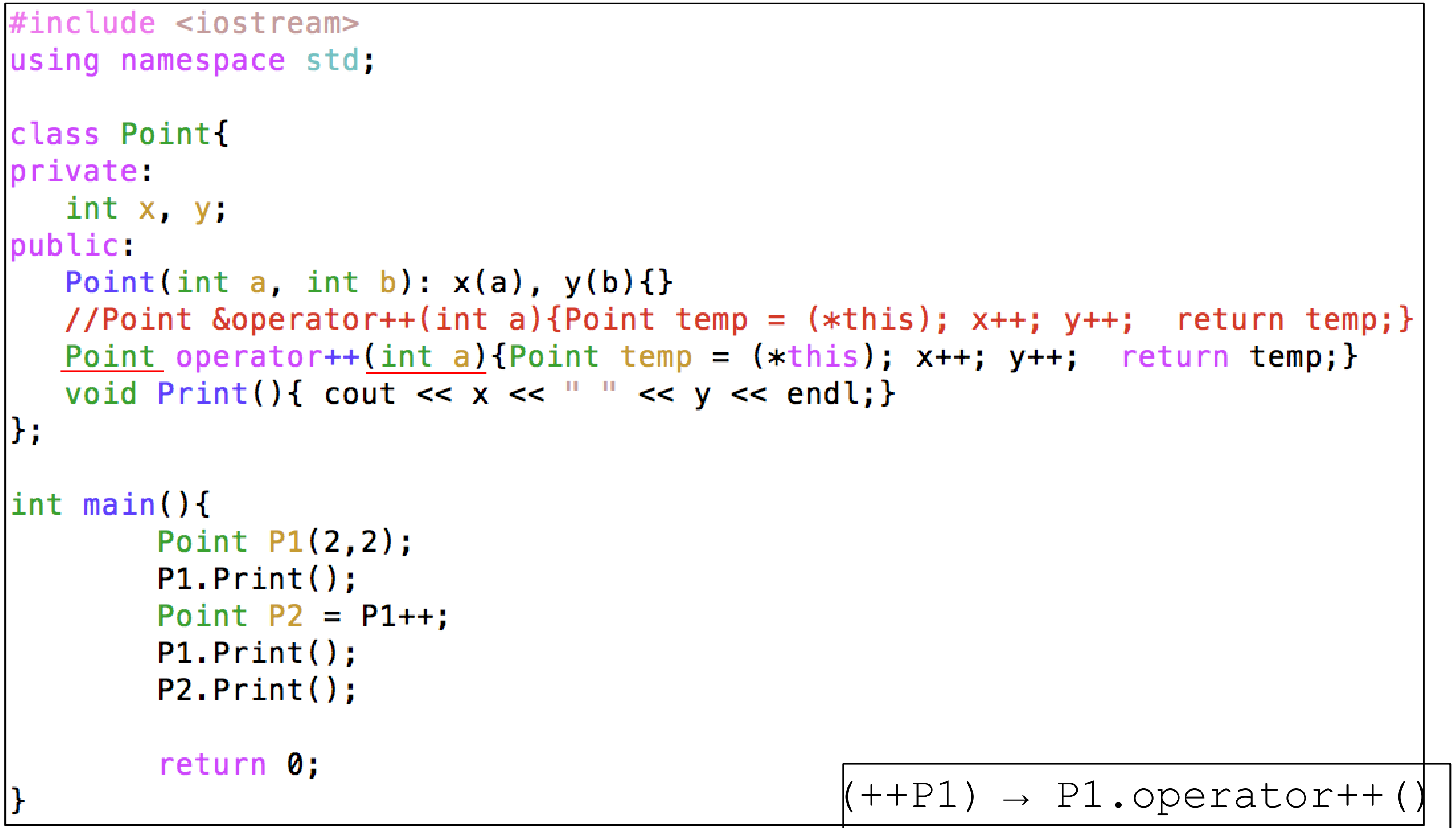
Operator Overloading: Increment Operator



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

class Point{
private:
   int x, y;
public:
   Point(int a, int b): x(a), y(b){}
   Point &operator++(){x++; y++;  return *this;}
   void Print(){ cout << x << " " << y << endl;}
};

int main(){
        Point P1(2,2);
        P1.Print();
        Point P2 = ++P1;
        P1.Print();
        (++P1).Print();

        return 0;
}
```

(++P1)→P1.operator++()

Operator Overloading: Increment Operator



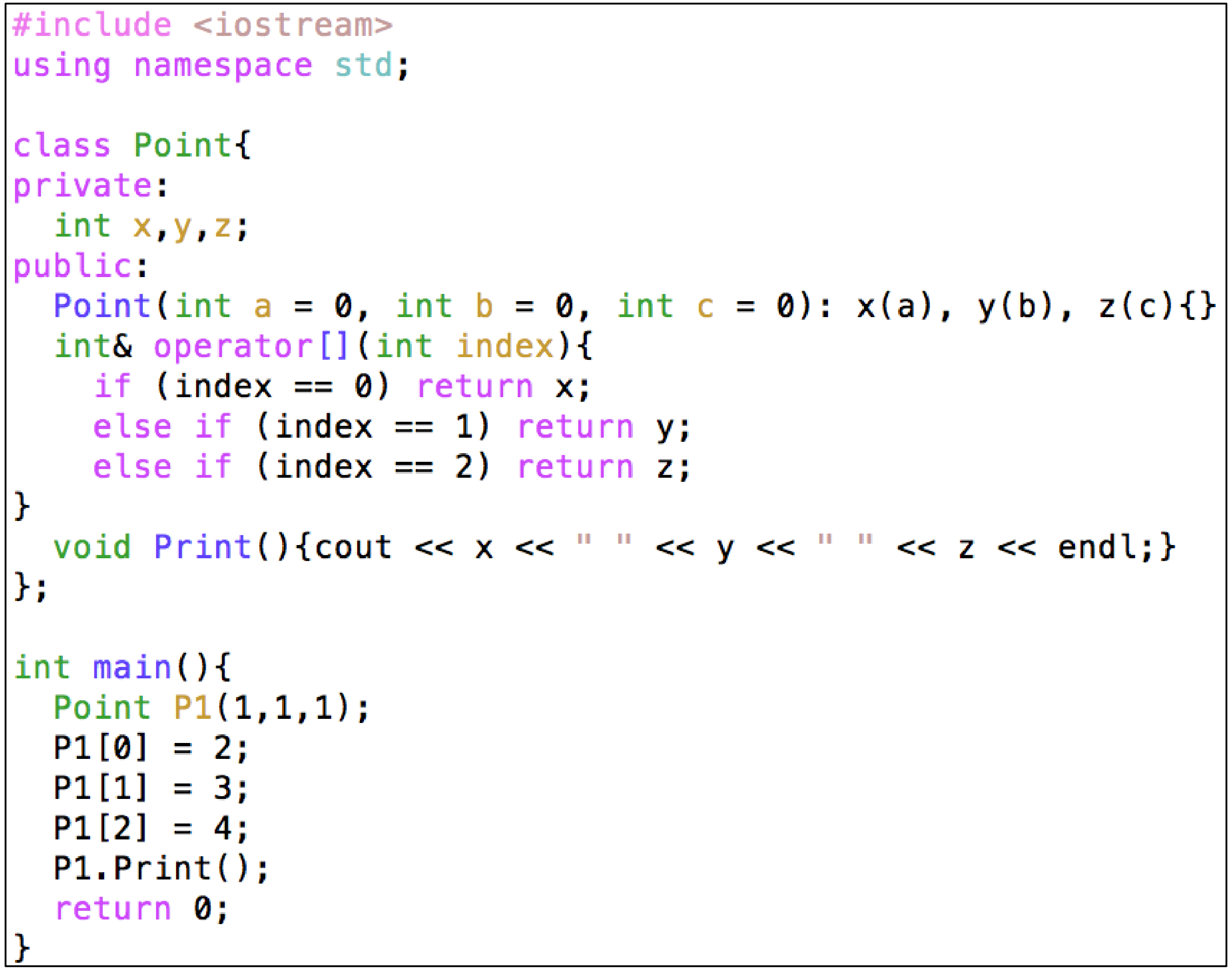
```
#include <iostream>
using namespace std;

class Point{
private:
   int x, y;
public:
   Point(int a, int b): x(a), y(b){}
   //Point &operator++(int a){Point temp = (*this); x++; y++;  return temp;}
   Point operator++(int a){Point temp = (*this); x++; y++;  return temp;}
   void Print(){ cout << x << " " << y << endl;}
};

int main(){
        Point P1(2,2);
        P1.Print();
        Point P2 = P1++;
        P1.Print();
        P2.Print();

        return 0;
}
```

(++P1) → P1.operator++()

(P1++) → P1.operator++(0)

Reference: https://www.learncpp.com/cpp-tutorial/97-overloading-the-increment-and-decrement-operators/

Operator Overloading: []



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

class Point{
private:
  int x,y,z;
public:
  Point(int a = 0, int b = 0, int c = 0): x(a), y(b), z(c){}
  int& operator[](int index){
    if (index == 0) return x;
    else if (index == 1) return y;
    else if (index == 2) return z;
}
  void Print(){cout << x << " " << y << " " << z << endl;}
};

int main(){
  Point P1(1,1,1);
  P1[0] = 2;
  P1[1] = 3;
  P1[2] = 4;
  P1.Print();
  return 0;
}
```

Operator Overloading: Summary

```
class A {                                  // A a0, a1;
  A& operator =(const A& a);               // a0 = a1;
  A operator +(const A& a) const;          // a0 + a1
  A operator +() const;                    // +a0
  A& operator +=(const A& a);              // a0 += a1;
  A& operator ++();                        // ++a0
  A operator ++(int);                     // a0++
};

A operator +(const A& a0, const A& a1);    // a0 + a1
A operator +(const A& a0);                 // +a0
A& operator +=(A& a0, const A& a1);        // a0 += a1;
A& operator ++(A& a0);                     // ++a0
A operator ++(A& a0, int);                // a0++

std::ostream& operator <<(std::ostream& out, const A& a);    // cout << a0;
```

Operator Overloading: Summary

- In general, an operator whose result is ...
  - New value: Returns the new value by value
    - – e.g. +, -, ...
  - Existing value, but modified: Returns a reference to the modified value.
    - – e.g. =, +=, ...

Operator Overloading: Summary

- The C++ language rarely puts constraints on operator overloading such as
  - what the overloaded operators do
  - what should be the return type

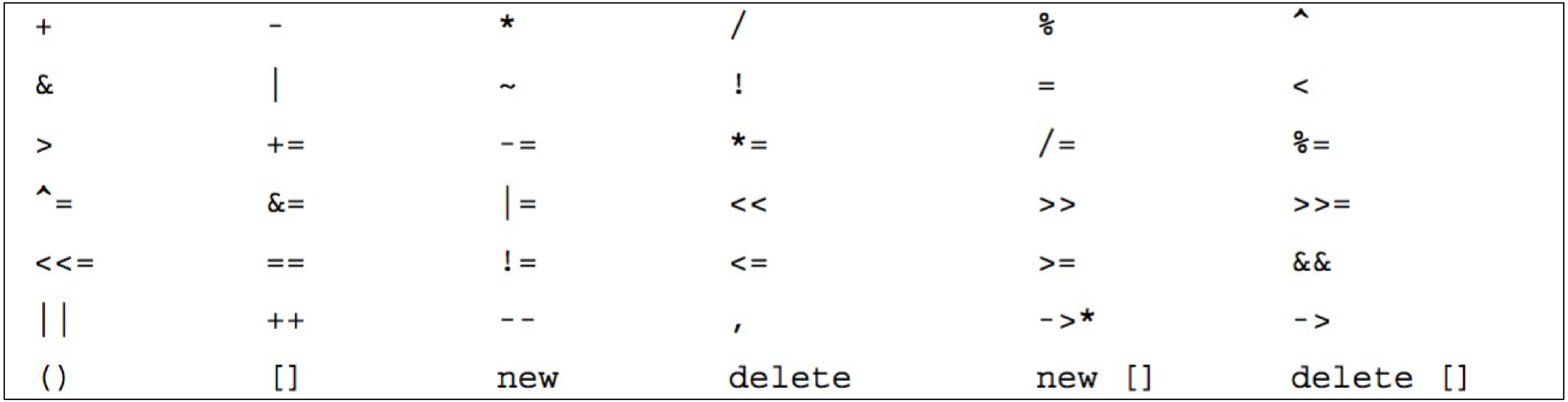
- But in general, overloaded operators are expected to behave as similar as possible to the built-in operators:
  - operator+ is expected to add, rather than multiply its arguments,
  - operator= is expected to assign

- The return types are limited by the expressions in which the operator is expected to be used:
  - for example, assignment operators return by reference to make it possible to write a = b = c = d, because the built-in operators allow that.

Operator Overloading: Summary

- Most commonly overloaded operators are
  - Arithmetic operators : +, -, *, / …
  - Assignment operators : =, +=, -=, *= …
  - Comparison operators : <, >, <=, >=, ==, != …
  - For array or containers : [], () …
  - Rarely : ->, new, delete, ...
- Operator overloading must be used very carefully, since it can hamper the readability seriously.

Operator That Can Be Overloaded



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

Operator That Cannot Be Overloaded

- Member access operator: **.** (a.b)

- Pointer to member access operator **.*** (a.*b)

- Name resolution operator: **::** (std::cout)

- Ternary conditional operator: **?:** ((a>b) ? 1 : 0)

- Macro operator: #, ##