Object Oriented Programming by C++

Software Component (2/2)

More about Class and Object

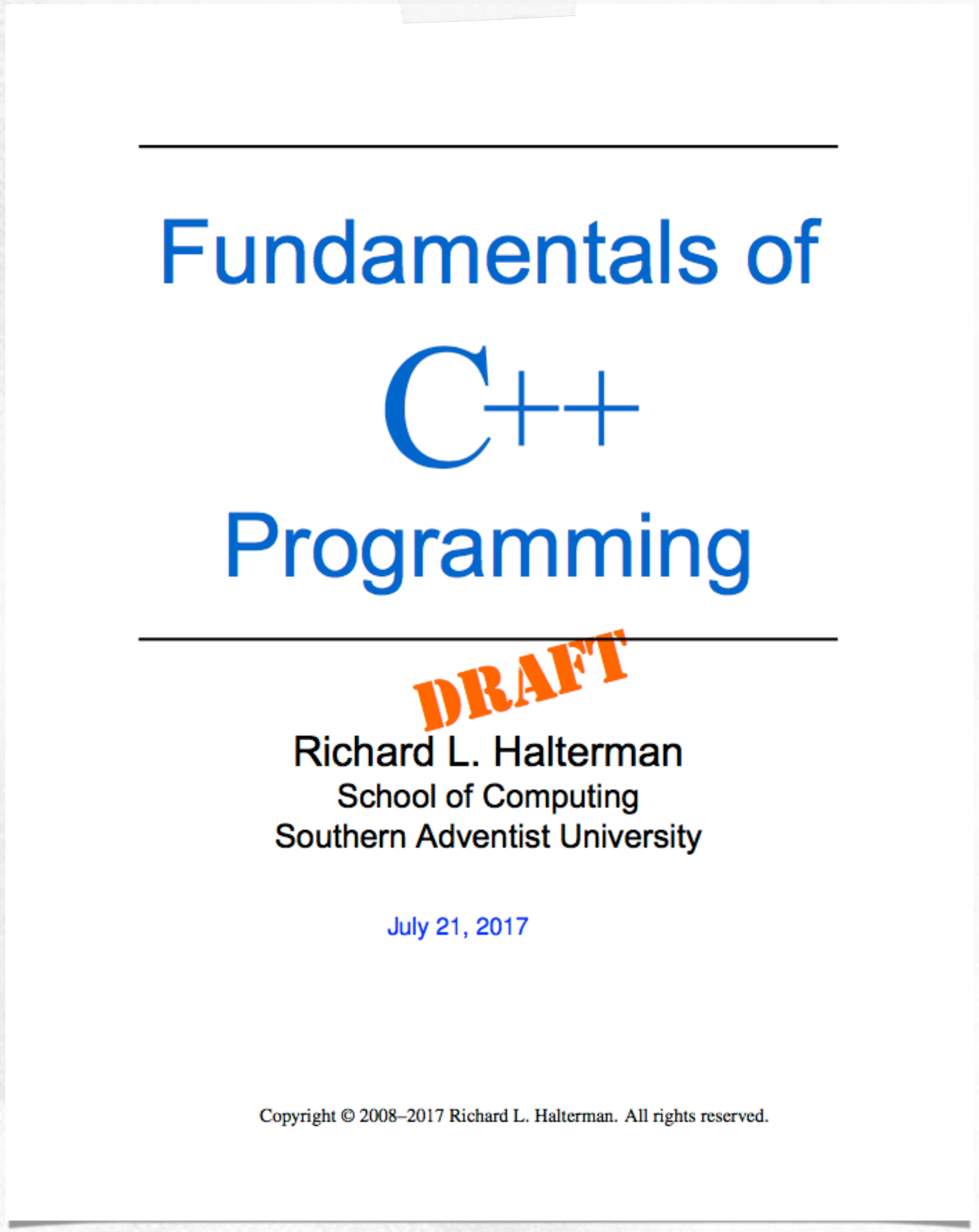
2017. 8.

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr
Web: http://mobilelab.khu.ac.kr/

Textbook & Copyright

- Textbook: http://python.cs.southern.edu/cppbook/progcpp.pdf
- Sample Codes: https://github.com/halterman/CppBook-SourceCode



Fundamentals of

C++

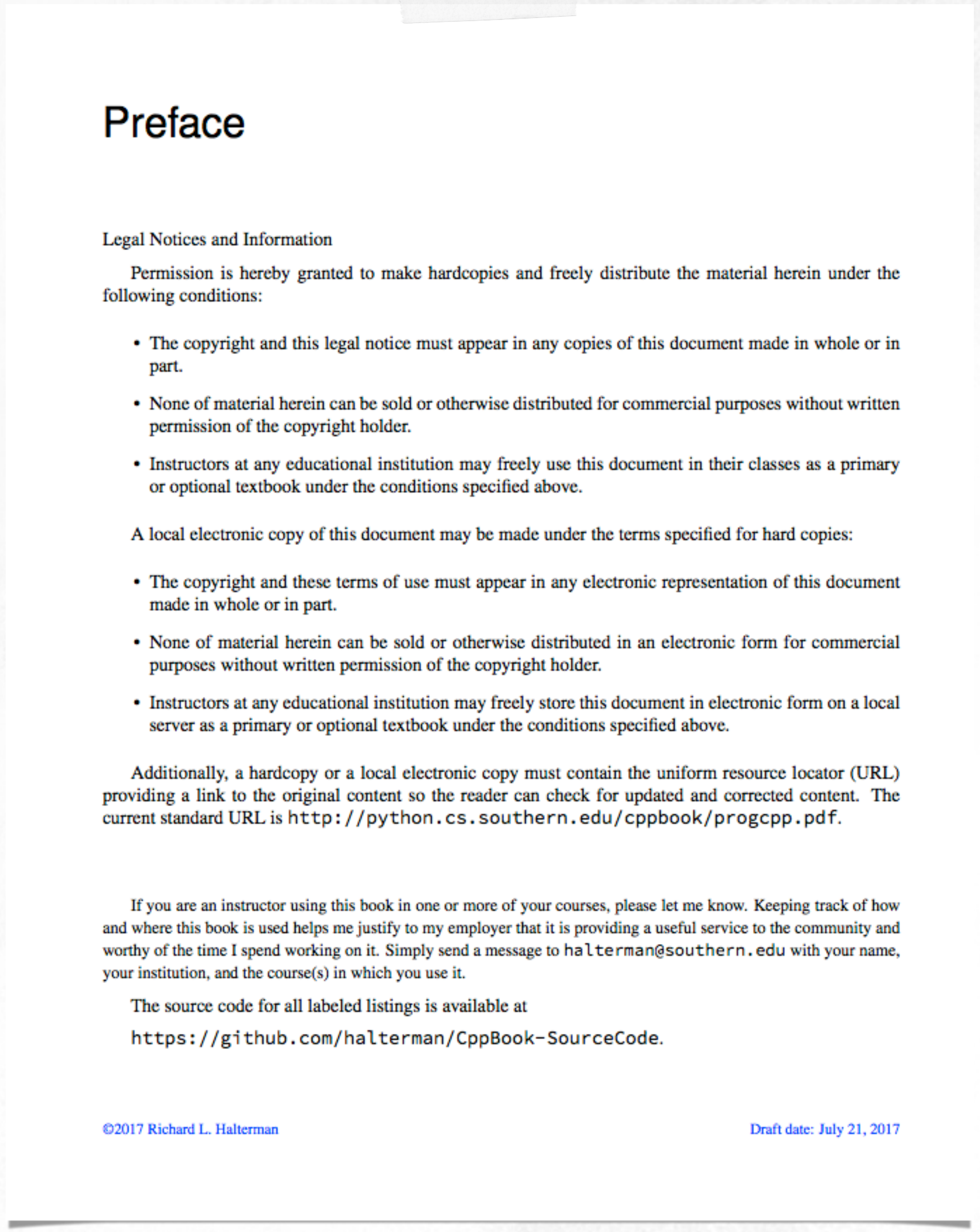
Programming

DRAFT

Richard L. Halterman
School of Computing
Southern Adventist University

July 21, 2017

Copyright © 2008–2017 Richard L. Halterman. All rights reserved.



Preface

Legal Notices and Information

Permission is hereby granted to make hardcopies and freely distribute the material herein under the following conditions:

- The copyright and this legal notice must appear in any copies of this document made in whole or in part.
- None of material herein can be sold or otherwise distributed for commercial purposes without written permission of the copyright holder.
- Instructors at any educational institution may freely use this document in their classes as a primary or optional textbook under the conditions specified above.

A local electronic copy of this document may be made under the terms specified for hard copies:

- The copyright and these terms of use must appear in any electronic representation of this document made in whole or in part.
- None of material herein can be sold or otherwise distributed in an electronic form for commercial purposes without written permission of the copyright holder.
- Instructors at any educational institution may freely store this document in electronic form on a local server as a primary or optional textbook under the conditions specified above.

Additionally, a hardcopy or a local electronic copy must contain the uniform resource locator (URL) providing a link to the original content so the reader can check for updated and corrected content. The current standard URL is `http://python.cs.southern.edu/cppbook/progcpp.pdf`.

If you are an instructor using this book in one or more of your courses, please let me know. Keeping track of how and where this book is used helps me justify to my employer that it is providing a useful service to the community and worthy of the time I spend working on it. Simply send a message to `halterman@southern.edu` with your name, your institution, and the course(s) in which you use it.

The source code for all labeled listings is available at

`https://github.com/halterman/CppBook-SourceCode`.

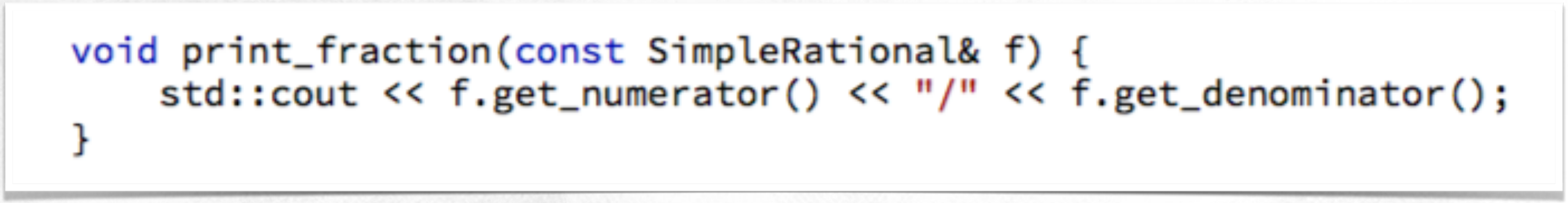
©2017 Richard L. Halterman   Draft date: July 21, 2017

Contents

Passing Object

Use 'Pass by Reference' & `const`

- For Passing Object and Vector
  - C++'s default "pass by value" causes large memory copy, and reduce the program execution performance
  - ***Efficiency of pass by reference***: The function has access to the caller's object via its location in memory. Pass by reference sends a single memory address to the function. There is no need to copy the object's data into the function.
  - ***Safety of pass by value (if required)***: The function cannot modify the caller's actual parameter since the formal parameter is declared `const`.

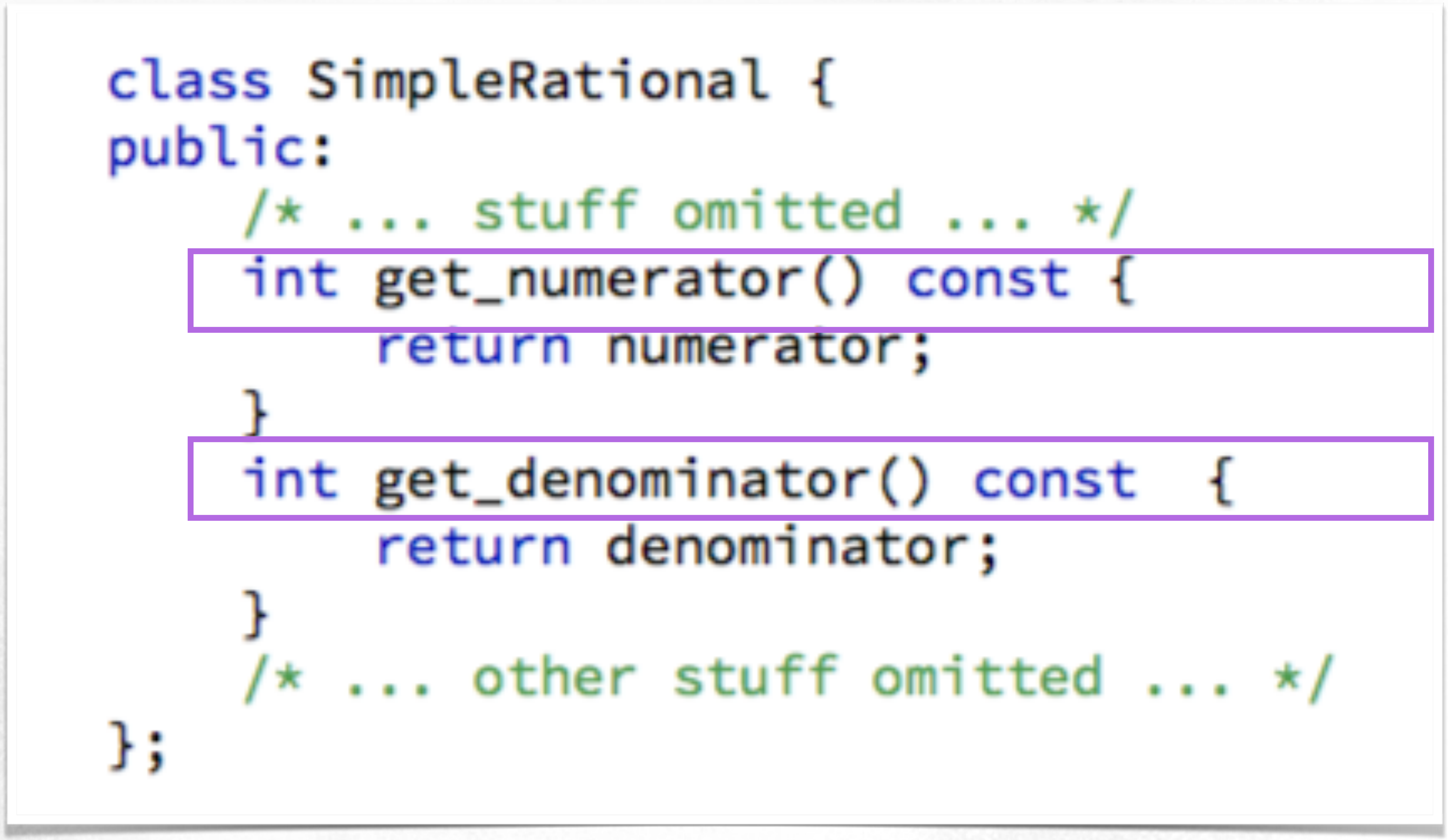


```
void print_fraction(const SimpleRational& f) {
    std::cout << f.get_numerator() << "/" << f.get_denominator();
}
```

Passing Object

Use 'Pass by Reference' & Const

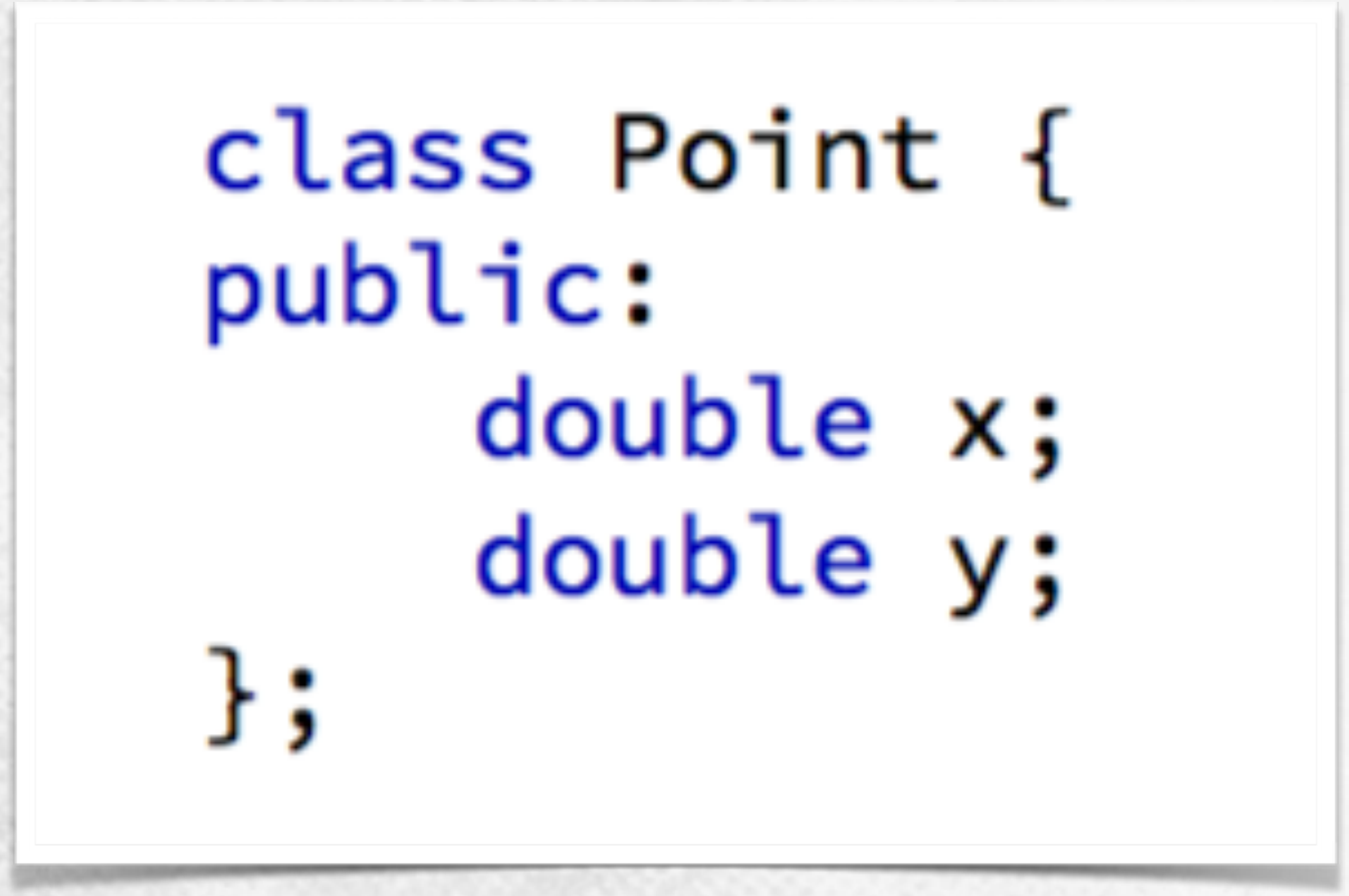
- If your own object makes error with ***const*** statement:
  - You should declare member function (which just retrieves member data, and not changes its value) type as ***const***
  - READ "15.4 const Methods" in Textbook



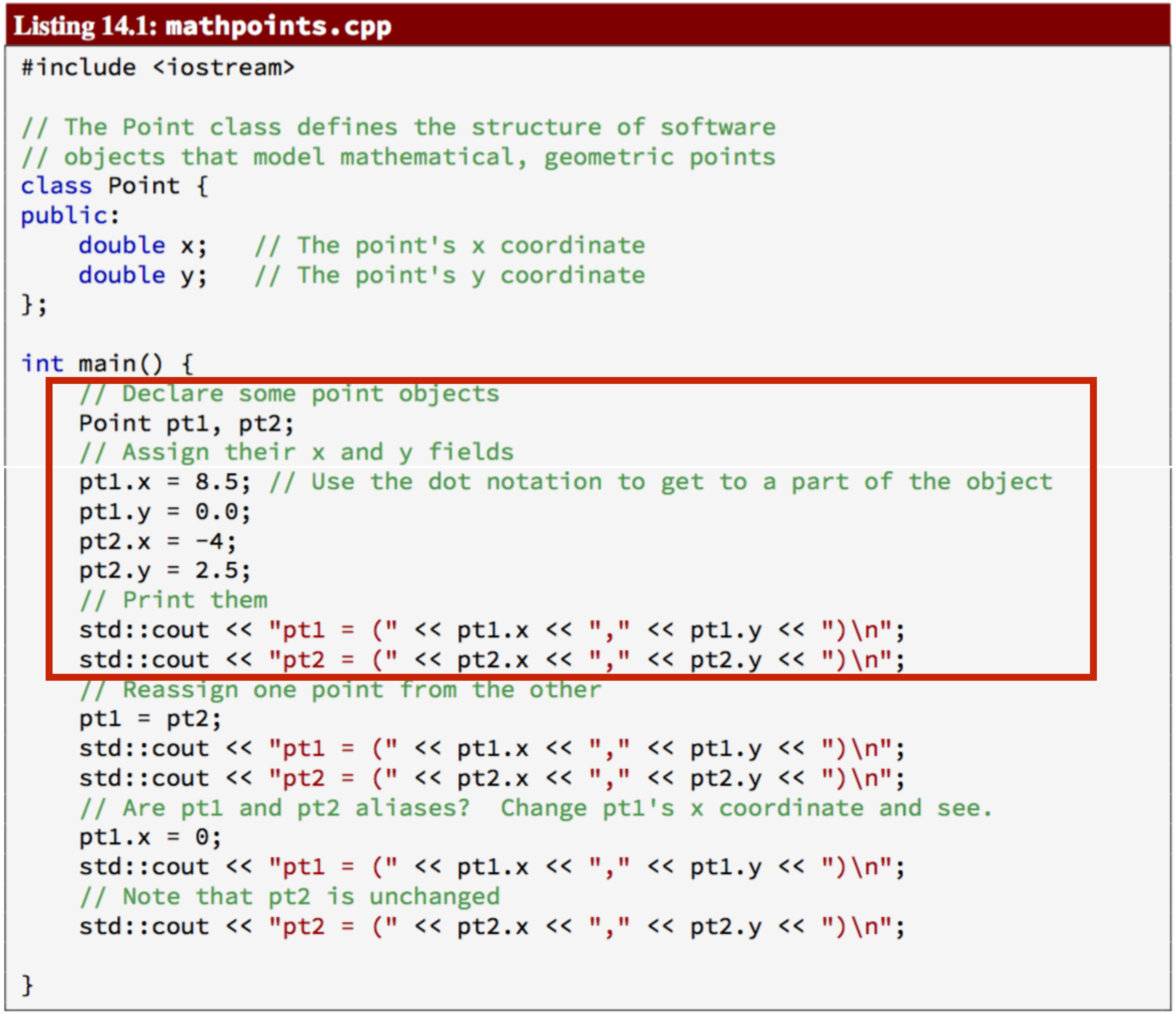
```
class SimpleRational {
public:
    /* ... stuff omitted ... */
    int get_numerator() const {
        return numerator;
    }
    int get_denominator() const  {
        return denominator;
    }
    /* ... other stuff omitted ... */
};
```

Pointers to Objects

Recall “Point Class”



```cpp
class Point {
public:
     double x;
     double y;
};
```



Listing 14.1: mathpoints.cpp

```cpp
#include <iostream>

// The Point class defines the structure of software
// objects that model mathematical, geometric points
class Point {
public:
    double x;    // The point's x coordinate
    double y;    // The point's y coordinate
};

int main() {
    // Declare some point objects
    Point pt1, pt2;
    // Assign their x and y fields
    pt1.x = 8.5; // Use the dot notation to get to a part of the object
    pt1.y = 0.0;
    pt2.x = -4;
    pt2.y = 2.5;
    // Print them
    std::cout << "pt1 = (" << pt1.x << "," << pt1.y << ")\n";
    std::cout << "pt2 = (" << pt2.x << "," << pt2.y << ")\n";
    // Reassign one point from the other
    pt1 = pt2;
    std::cout << "pt1 = (" << pt1.x << "," << pt1.y << ")\n";
    std::cout << "pt2 = (" << pt2.x << "," << pt2.y << ")\n";
    // Are pt1 and pt2 aliases?  Change pt1's x coordinate and see.
    pt1.x = 0;
    std::cout << "pt1 = (" << pt1.x << "," << pt1.y << ")\n";
    // Note that pt2 is unchanged
    std::cout << "pt2 = (" << pt2.x << "," << pt2.y << ")\n";

}
```

Applying "Pointer to Object"

- Same rules:
  - Pointer Definition

    Point pt1;

    Point* pt2;
  - Operator '&'

    pt2 = &pt1;
  - Operator '*'

    (*pt2).x = 1.0

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

class Point {
public:
    double x;
    double y;
};

int main()
{
    Point pt1;
    Point *pt2;

    pt2 = &pt1;

    pt1.x = 100.0;
    pt1.y = 200.0;

    cout << (*pt2).x << ":" << (*pt2).y;
}
```

New "Pointer to Object" Operator

- Operator '->'
  - Same meaning:

(*pt2).x = 1.0

and

pt2->x = 1.0

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

class Point {
public:
    double x;
    double y;
};

int main()
{
    Point pt1;
    Point *pt2;

    pt2 = &pt1;

    pt1.x = 100.0;
    pt1.y = 200.0;

    cout << pt2->x << ":" << pt2->y;
}
```

Pointers to Objects

Using new and delete operator

- Same rules:
  - Dynamic allocation

    Point* pt2;

    pt2 = new Point;

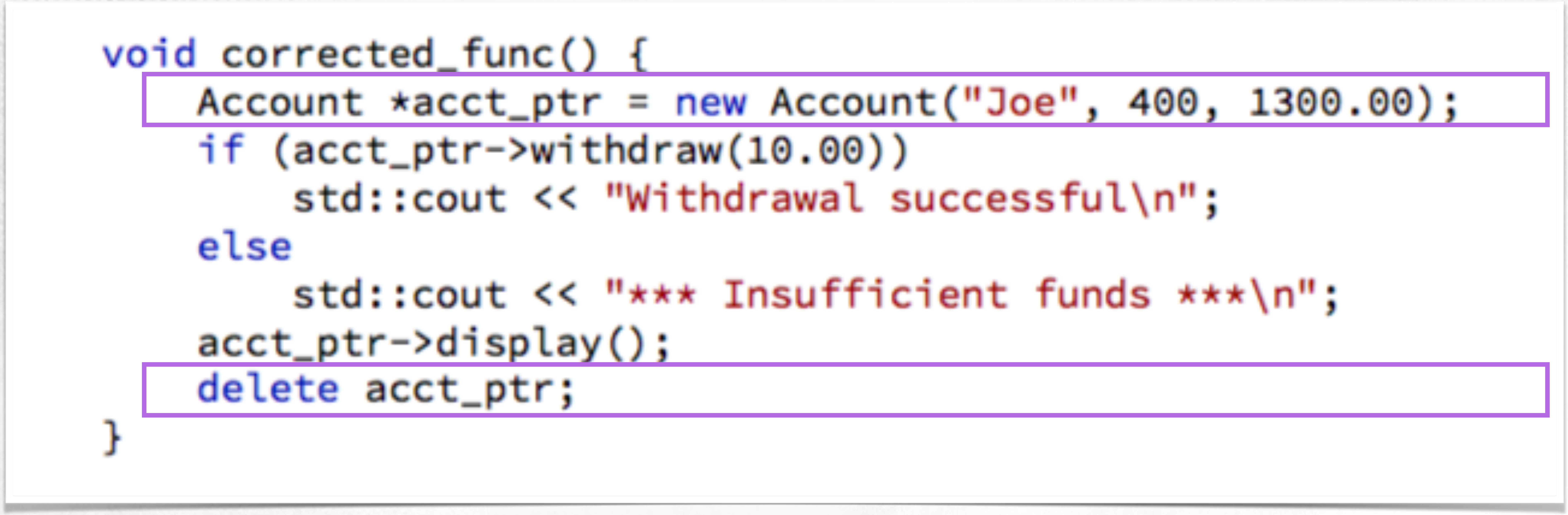
  - Deallocation

    delete pt2;

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

class Point {
public:
    double x;
    double y;
};

int main()
{
    Point *pt2;

    pt2 = new Point;

    pt2->x = 100.0;
    pt2->y = 200.0;

    cout << pt2->x << ":" << pt2->y;

    delete pt2;
}
```

NOTE for Using new and delete operator

- Dynamically allocated ***local variables*** should be removed when return.

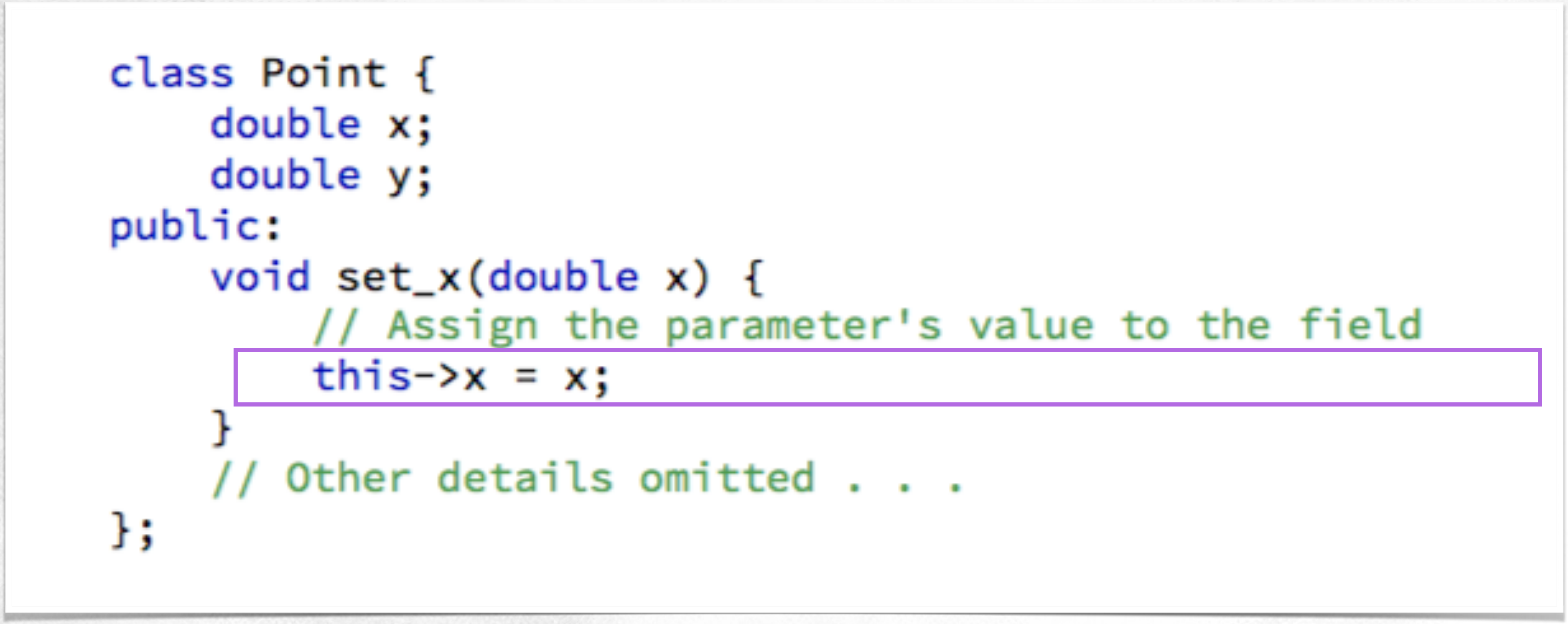


```
void corrected_func() {
    Account *acct_ptr = new Account("Joe", 400, 1300.00);
    if (acct_ptr->withdraw(10.00))
        std::cout << "Withdrawal successful\n";
    else
        std::cout << "*** Insufficient funds ***\n";
    acct_ptr->display();
    delete acct_ptr;
}
```

Pointers to Objects

this Pointer

- this Pointer is " ***the address of the object itself*** ":
  - Hidden local variable (automatically generated)
  - Used to avoid parameter confusion, specific address pointing, etc.



```
class Point {
    double x;
    double y;
public:
    void set_x(double x) {
        // Assign the parameter's value to the field
        this->x = x;
    }
    // Other details omitted . . .
};
```

Function Overloading

- C++ allows you to specify ***more than one definition for a function name or an operator in the same scope***, which is called function overloading and operator overloading respectively.
- An overloaded declaration is a declaration that is declared with ***the same name as a previously declared declaration in the same scope***, except that both declarations have different arguments and obviously different definition (implementation).

Operator Overloading

- Do you want

$$(x_1, y_1) + (x_2, y_2) = (x_1 + y_1, x_2 + y_2)$$

  using ***Point*** class?

- Operator '+' can be defined as: (example: "*Non-member function scenario*")

```
Point operator+(const Point& p1, const Point& p2) {
     Point result;
     result.x = p1.x + p2.x;
     result.y = p1.y + p2.y;
     return result;
}
```

- Several combinations are possible - ***Case by Case***

Code Example 1

Put it all Together !! (1/2)

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

class Point {
   double x;
   double y;

public:
   Point();
   Point(int x, int y);
   void setPoint(int x, int y);
   int getX(void) const;
   int getY(void) const;
   Point operator+(const Point& point);
   Point& operator=(const Point& point);
};

Point::Point()
{
   x = y = 0;
}

Point::Point(int x, int y)
{
   this->x = x;
   this->y = y;
}

void Point::setPoint(int x, int y)
{
   this->x = x;
   this->y = y;
}
```

```
int Point::getX (void) const
{
   return this->x;
}

int Point::getY (void) const
{
   return this->y;
}

Point Point::operator+(const Point& point)
{
   Point result(this->x + point.getX(), this->y + point.getY());
   return result;
}

Point& Point::operator=(const Point& point)
{
   this->x = point.getX();
   this->y = point.getY();
   return *this;
}

std::ostream& operator<<(std::ostream& os, const Point& point)
{
   return os << "(" << point.getX() << "," << point.getY() << ")";
}
```

Code Example 1

Put it all Together !! (2/2)

```
int main()
{
   Point *pP1, *pP2;

   pP1 = new Point;
   pP2 = new Point( 1, 2 );

   pP1->setPoint( 10, 20 );

   *pP2 = *pP1 + *pP2;

   cout << "[X:" << pP1->getX() << "]" << "[Y:" << pP1->getY() << "]" << endl;
   cout << *pP2 << endl;

   delete pP1;
   delete pP2;
}
```

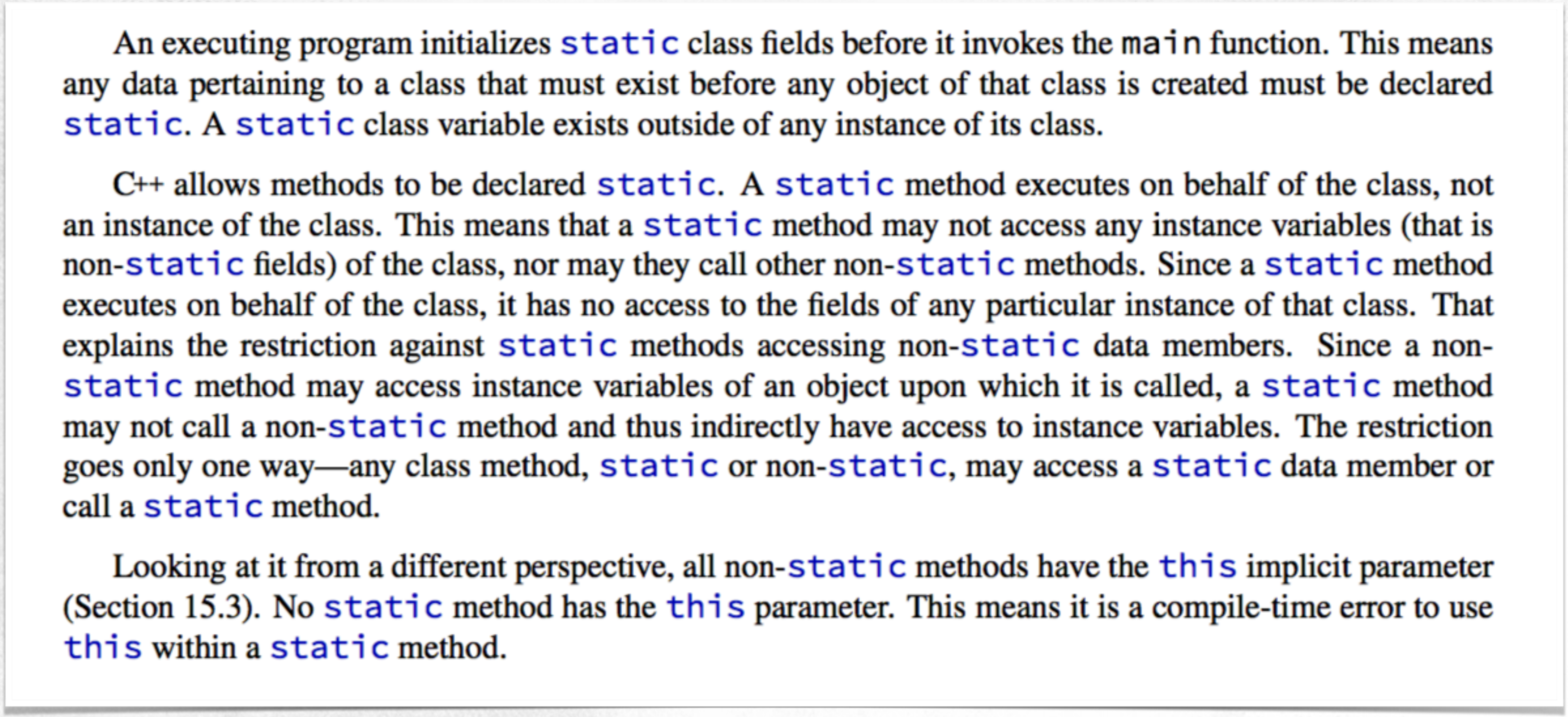
```
[X:10][Y:20]
(11,22)
```

static Member Data and Functions

- Sometimes it is convenient to have variables or ***constants that all objects of a class share***.
  - Global variables and constants certainly will work, but globals are not tied to any particular class
  - C++ uses the **`static`** keyword within a class to specify that all objects of that class share a field

```
class Point {
   double x;
   double y;
   static int countCreatedObjects;

public:
   Point();
   Point(int x, int y);
   void setPoint(int x, int y);
   int getX(void) const;
   int getY(void) const;
   static int getCreatedObject(void);
   Point& operator+(const Point& point);
   Point& operator=(const Point& point);
};
```

`static` Member Data and Functions



An executing program initializes `static` class fields before it invokes the `main` function. This means any data pertaining to a class that must exist before any object of that class is created must be declared `static`. A `static` class variable exists outside of any instance of its class.

C++ allows methods to be declared `static`. A `static` method executes on behalf of the class, not an instance of the class. This means that a `static` method may not access any instance variables (that is non-`static` fields) of the class, nor may they call other non-`static` methods. Since a `static` method executes on behalf of the class, it has no access to the fields of any particular instance of that class. That explains the restriction against `static` methods accessing non-`static` data members. Since a non-`static` method may access instance variables of an object upon which it is called, a `static` method may not call a non-`static` method and thus indirectly have access to instance variables. The restriction goes only one way—any class method, `static` or non-`static`, may access a `static` data member or call a `static` method.

Looking at it from a different perspective, all non-`static` methods have the `this` implicit parameter (Section 15.3). No `static` method has the `this` parameter. This means it is a compile-time error to use `this` within a `static` method.

Code Example 2

Enhancement using static (1/2)

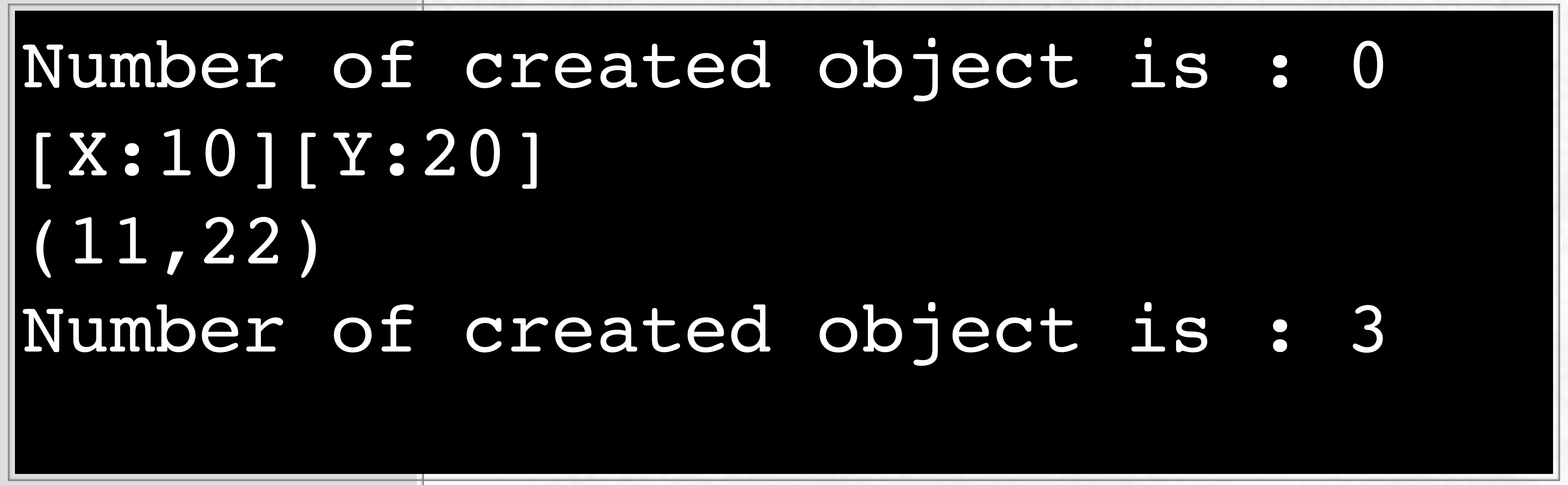
```
#include <iostream>
using namespace std;

class Point {
   double x;
   double y;
   static int countCreatedObjects;

public:
   Point();
   Point(int x, int y);
   void setPoint(int x, int y);
   int getX(void) const;
   int getY(void) const;
   static int getCreatedObject(void);
   Point operator+(const Point& point);
   Point& operator=(const Point& point);
};

// Initialize the static variables.
int Point::countCreatedObjects = 0;

Point::Point()
{
   x = y = 0;
   countCreatedObjects++;
}

Point::Point(int x, int y)
{
   this->x = x;
   this->y = y;
   countCreatedObjects++;
}
```

```
void Point::setPoint(int x, int y)
{
   this->x = x;
   this->y = y;
}

int Point::getX (void) const
{
   return this->x;
}

int Point::getY (void) const
{
   return this->y;
}

int Point::getCreatedObject(void)
{
   return countCreatedObjects;
}

Point Point::operator+(const Point& point)
{
   Point result(this->x + point.getX(), this->y + point.getY());
   return result;
}

Point& Point::operator=(const Point& point)
{
   this->x = point.getX();
   this->y = point.getY();
   return *this;
}
```

Code Example 2

Enhancement using static (2/2)

```
std::ostream& operator<<(std::ostream& os, const Point& point)
{
   return os << "(" << point.getX() << "," << point.getY() << ")";
}

int main()
{
   Point *pP1, *pP2;

   cout << "Number of created object is : " << Point::getCreatedObject() << endl;

   pP1 = new Point;
   pP2 = new Point( 1, 2 );

   pP1->setPoint( 10, 20 );

   *pP2 = *pP1 + *pP2;

   cout << "[X:" << pP1->getX() << "]" << "[Y:" << pP1->getY() << "]" << endl;
   cout << *pP2 << endl;

   cout << "Number of created object is : " << Point::getCreatedObject() << endl;

   delete pP1;
   delete pP2;
}
```



```
Number of created object is : 0
[X:10][Y:20]
(11,22)
Number of created object is : 3
```

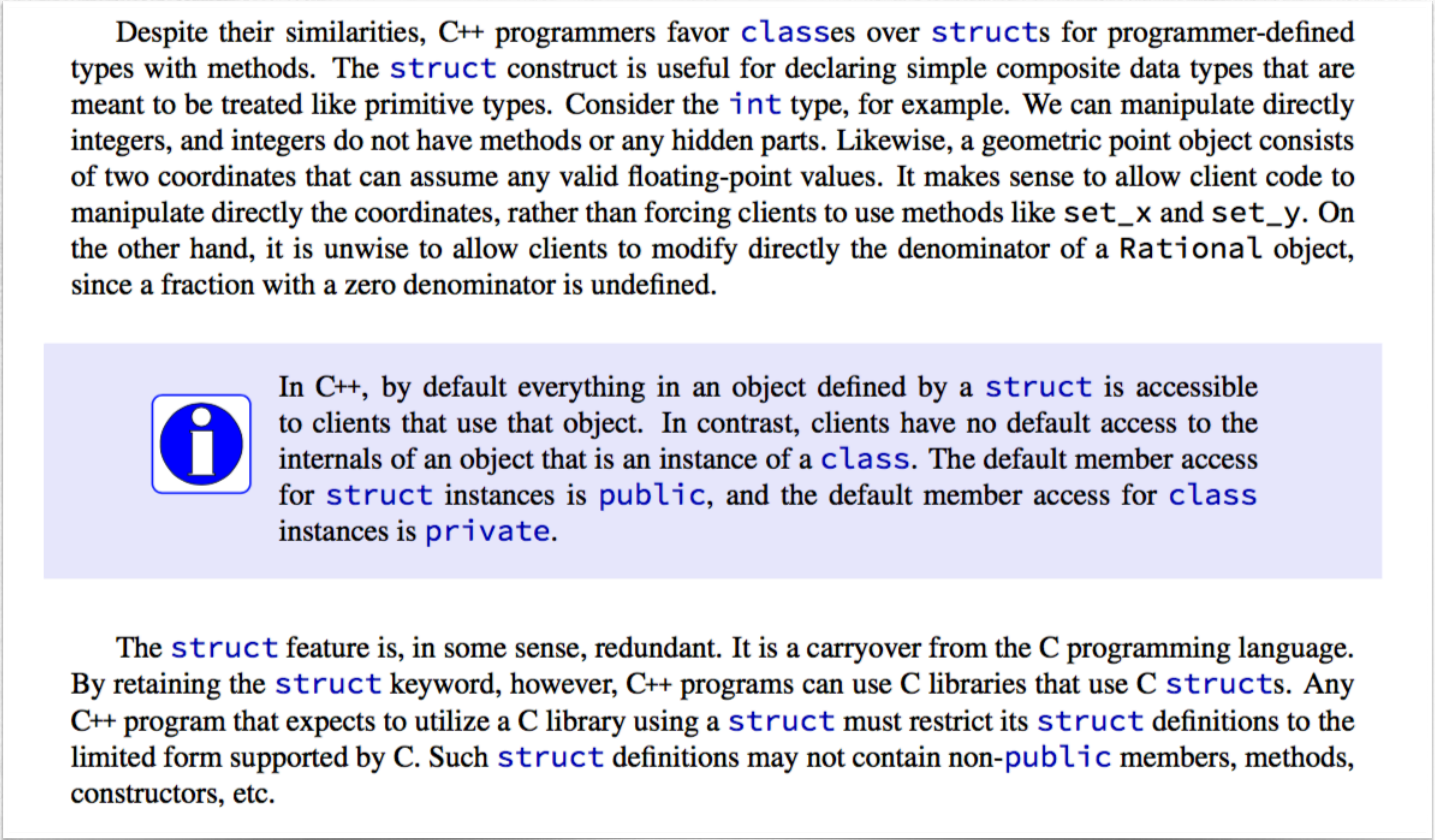
Classes vs. Structures

- Structure (`struct`) is derived from C language
- Similar statement with Class definition
- BUT EVERY MEMBER IS ***PUBLIC*** BY DEFAULT !!!!
- Except that:
  - Structure can contain methods and constructors
  - You can apply the private and public labels as in a class

```
class Point {
 public:
    double x;
    double y;
};
```

```
struct Point {
    double x;
    double y;
};
```

Why we use Class in C++?



Despite their similarities, C++ programmers favor `class`es over `struct`s for programmer-defined types with methods. The `struct` construct is useful for declaring simple composite data types that are meant to be treated like primitive types. Consider the `int` type, for example. We can manipulate directly integers, and integers do not have methods or any hidden parts. Likewise, a geometric point object consists of two coordinates that can assume any valid floating-point values. It makes sense to allow client code to manipulate directly the coordinates, rather than forcing clients to use methods like `set_x` and `set_y`. On the other hand, it is unwise to allow clients to modify directly the denominator of a `Rational` object, since a fraction with a zero denominator is undefined.

In C++, by default everything in an object defined by a `struct` is accessible to clients that use that object. In contrast, clients have no default access to the internals of an object that is an instance of a `class`. The default member access for `struct` instances is `public`, and the default member access for `class` instances is `private`.

The `struct` feature is, in some sense, redundant. It is a carryover from the C programming language. By retaining the `struct` keyword, however, C++ programs can use C libraries that use C `struct`s. Any C++ program that expects to utilize a C library using a `struct` must restrict its `struct` definitions to the limited form supported by C. Such `struct` definitions may not contain non-`public` members, methods, constructors, etc.

Friend Function

- At times it can be advantageous to design a class that **grants special access** to some precisely ***specified functions or classes of objects outside of the class***.

Code Example 3

Modification using friend function (1/2)

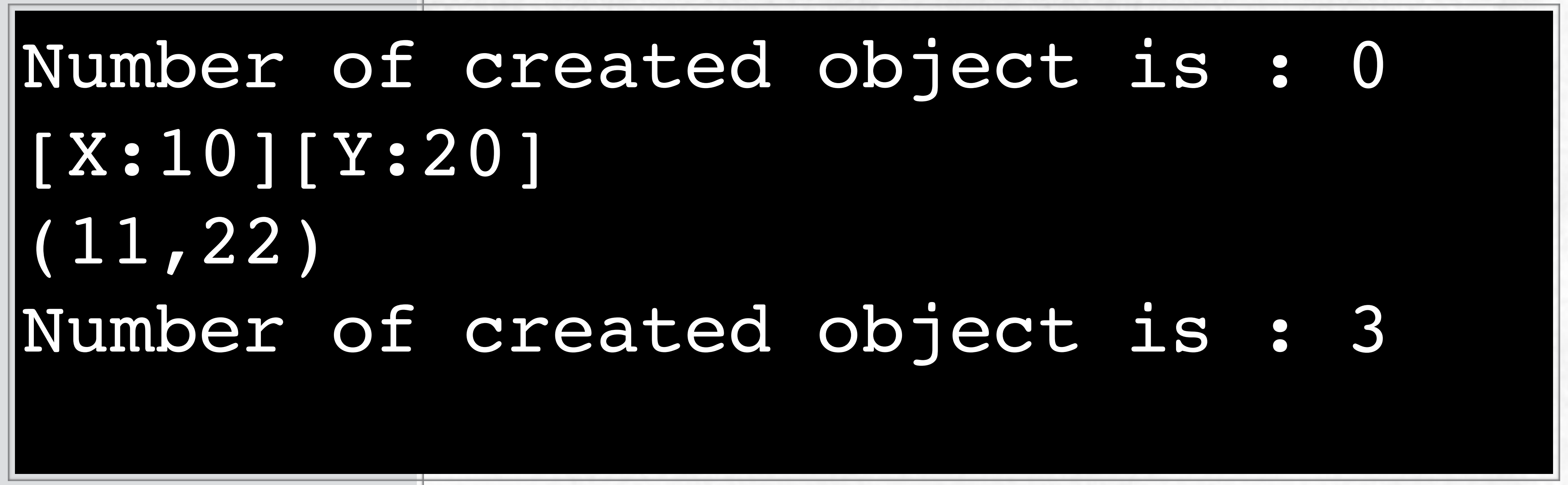
```
#include <iostream>
using namespace std;

class Point {
   double x;
   double y;
   static int countCreatedObjects;

public:
   Point();
   Point(int x, int y);
   void setPoint(int x, int y);
   int getX(void) const;
   int getY(void) const;
   static int getCreatedObject(void);
   Point operator+(const Point& point);
   Point& operator=(const Point& point);
   friend std::ostream& operator<<(std::ostream& os, const Point& point);
};

// Initialize the static variables.
int Point::countCreatedObjects = 0;

Point::Point()
{
   x = y = 0;
   countCreatedObjects++;
}

Point::Point(int x, int y)
{
   this->x = x;
   this->y = y;
   countCreatedObjects++;
}
```

```
void Point::setPoint(int x, int y)
{
   this->x = x;
   this->y = y;
}

int Point::getX (void) const
{
   return this->x;
}

int Point::getY (void) const
{
   return this->y;
}

int Point::getCreatedObject(void)
{
   return countCreatedObjects;
}

Point Point::operator+(const Point& point)
{
   Point result(this->x + point.getX(),
                this->y + point.getY());
   return result;
}

Point& Point::operator=(const Point& point)
{
   this->x = point.getX();
   this->y = point.getY();
   return *this;
}
```

Code Example 3

Modification using friend function (2/2)

```
std::ostream& operator<<(std::ostream& os, const Point& point)
{
   return os << "(" << point.x << "," << point.y << ")";
}

int main()
{
   Point *pP1, *pP2;

   cout << "Number of created object is : " << Point::getCreatedObject() << endl;

   pP1 = new Point;
   pP2 = new Point( 1, 2 );

   pP1->setPoint( 10, 20 );

   *pP2 = *pP1 + *pP2;

   cout << "[X:" << pP1->getX() << "]" << "[Y:" << pP1->getY() << "]" << endl;
   cout << *pP2 << endl;

   cout << "Number of created object is : " << Point::getCreatedObject() << endl;

   delete pP1;
   delete pP2;
}
```



```
Number of created object is : 0
[X:10][Y:20]
(11,22)
Number of created object is : 3
```

Code Example 4

Modification using friend Class (1/2)

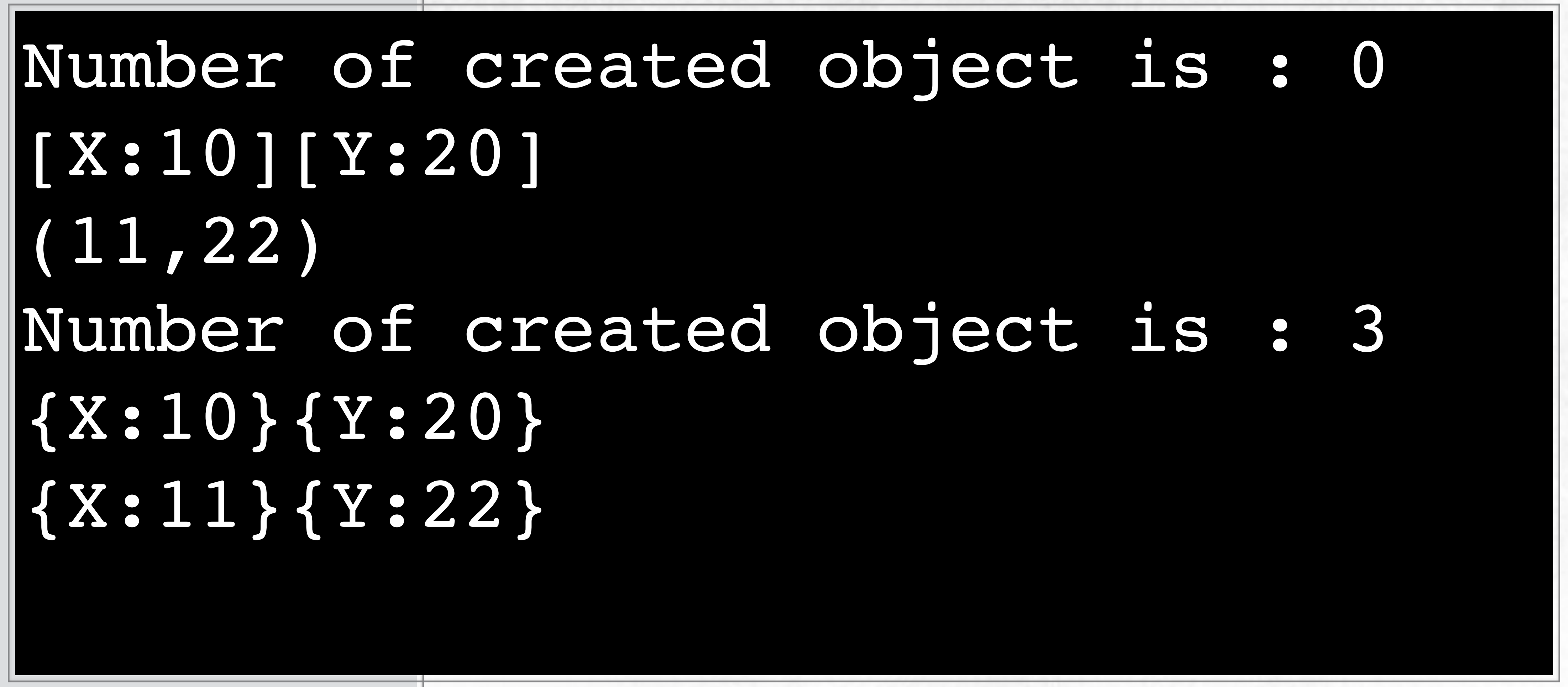
```
#include <iostream>
using namespace std;

class Point {
   double x;
   double y;
   static int countCreatedObjects;

public:
   Point();
   Point(int x, int y);
   void setPoint(int x, int y);
   int getX(void) const;
   int getY(void) const;
   static int getCreatedObject(void);
   Point operator+(const Point& point);
   Point& operator=(const Point& point);
   friend std::ostream& operator<<(std::ostream& os, const Point& point);
   friend class SpyPoint;
};

int Point::countCreatedObjects = 0; // Initialize the static variables.

Point::Point()
{
   x = y = 0;
   countCreatedObjects++;
}

Point::Point(int x, int y)
{
   this->x = x;
   this->y = y;
   countCreatedObjects++;
}
```

```
void Point::setPoint(int x, int y)
{
   this->x = x;
   this->y = y;
}

int Point::getX (void) const
{
   return this->x;
}

int Point::getY (void) const
{
   return this->y;
}

int Point::getCreatedObject(void)
{
   return countCreatedObjects;
}

Point Point::operator+(const Point& point)
{
   Point result(this->x + point.getX(),
                this->y + point.getY());
   return result;
}

Point& Point::operator=(const Point& point)
{
   this->x = point.getX();
   this->y = point.getY();
   return *this;
}
```

Code Example 4

Modification using friend Class (1/2)

```
std::ostream& operator<<(std::ostream& os, const Point& point)
{
   return os << "(" << point.x << "," << point.y << ")";
}

class SpyPoint {
public:
   void printPoint(const Point& point);
};

void SpyPoint::printPoint(const Point& point)
{
   cout << "{X:" << point.x << "}" << "{Y:" << point.y << "}" << endl;
}

int main()
{
   Point *pP1, *pP2;
   SpyPoint SP;

   cout << "Number of created object is : " << Point::getCreatedObject() << endl;

   pP1 = new Point;
   pP2 = new Point( 1, 2 );

   pP1->setPoint( 10, 20 );

   *pP2 = *pP1 + *pP2;

   cout << "[X:" << pP1->getX() << "]" << "[Y:" << pP1->getY() << "]" << endl;
   cout << *pP2 << endl;
   cout << "Number of created object is : " << Point::getCreatedObject() << endl;

   SP.printPoint(*pP1);
   SP.printPoint(*pP2);

   delete pP1;
   delete pP2;
}
```



```
Number of created object is : 0
[X:10][Y:20]
(11,22)
Number of created object is : 3
{X:10}{Y:20}
{X:11}{Y:22}
```

Destructor Function

- A destructor is a special member function that is called when the lifetime of an object ends. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.
- Destructor have no return type, input parameters, and ~*ClassName*() format
- For example:
  - Memory allocation (`new`), File open @ Constructor
  - Memory release (`delete`), File close @ Destructor

Code Example 5

Modification using Destructor (1/4)

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

class Point {
   double x;
   double y;
   static int countCreatedObjects;

public:
   Point();
   Point(int x, int y);
   ~Point();
   void setPoint(int x, int y);
   int getX(void) const;
   int getY(void) const;
   static int getCreatedObject(void);
   Point operator+(const Point& point);
   Point& operator=(const Point& point);
   friend std::ostream& operator<<(std::ostream& os, const Point& point);
   friend class SpyPoint;
};

int Point::countCreatedObjects = 0; // Initialize the static variables.

Point::Point()
{
   cout << "constructor Point() invoked..." << endl;
   x = y = 0;
   countCreatedObjects++;
}
```

Code Example 5

Modification using Destructor (2/4)

```
Point::Point(int x, int y)
{
   cout << "constructor Point(int,int) invoked..." << endl;
   this->x = x;
   this->y = y;
   countCreatedObjects++;
}

Point::~Point()
{
   cout << "( " << this->x << ":" << this->y << ") destructor invoked..." << endl;
}

void Point::setPoint(int x, int y)
{
   this->x = x;
   this->y = y;
}

int Point::getX (void) const
{
   return this->x;
}

int Point::getY (void) const
{
   return this->y;
}

int Point::getCreatedObject(void)
{
   return countCreatedObjects;
}
```

Code Example 5

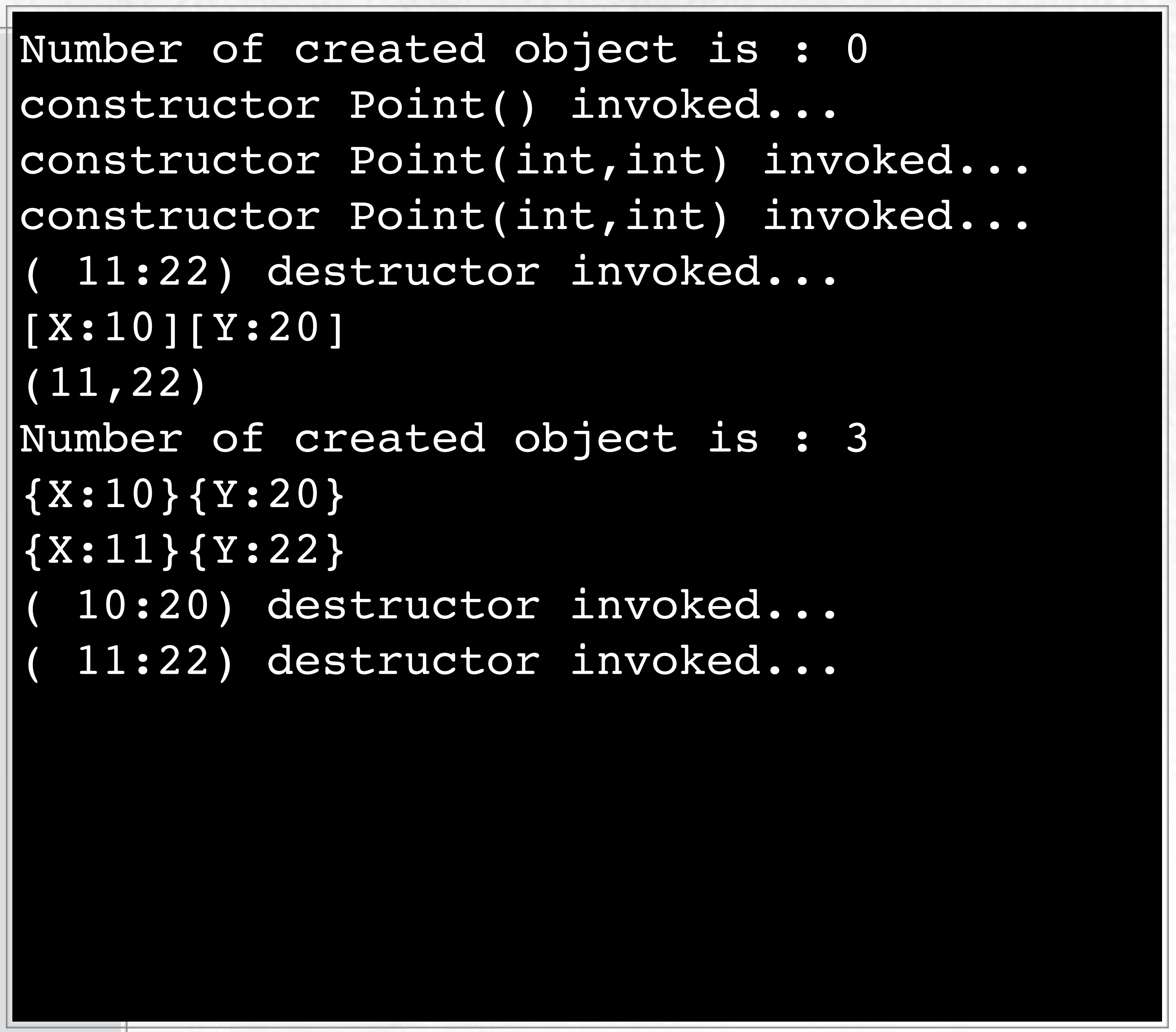
Modification using Destructor (3/4)

```
Point Point::operator+(const Point& point)
{
   Point result(this->x + point.getX(), this->y + point.getY());
   return result;
}

Point& Point::operator=(const Point& point)
{
   this->x = point.getX();
   this->y = point.getY();
   return *this;
}

std::ostream& operator<<(std::ostream& os, const Point& point)
{
   return os << "(" << point.x << "," << point.y << ")";
}

class SpyPoint {
public:
   void printPoint(const Point& point);
};

void SpyPoint::printPoint(const Point& point)
{
   cout << "{X:" << point.x << "}" << "{Y:" << point.y << "}" << endl;
}
```

Code Example 5

Modification using Destructor (4/4)

```
int main()
{
   Point *pP1, *pP2;
   SpyPoint SP;

   cout << "Number of created object is : " << Point::getCreatedObject() << endl;

   pP1 = new Point;
   pP2 = new Point( 1, 2 );

   pP1->setPoint( 10, 20 );

   *pP2 = *pP1 + *pP2;

   cout << "[X:" << pP1->getX() << "]" << "[Y:" << pP1->getY() << "]" << endl;
   cout << *pP2 << endl;
   cout << "Number of created object is : " << Point::getCreatedObject() << endl;

   SP.printPoint(*pP1);
   SP.printPoint(*pP2);

   delete pP1;
   delete pP2;
}
```



```
Number of created object is : 0
constructor Point() invoked...
constructor Point(int,int) invoked...
constructor Point(int,int) invoked...
( 11:22) destructor invoked...
[X:10][Y:20]
(11,22)
Number of created object is : 3
{X:10}{Y:20}
{X:11}{Y:22}
( 10:20) destructor invoked...
( 11:22) destructor invoked...
```

Recommendation

Code Review

Chapter 16

Building some Useful Classes

Object Oriented Programming by C++

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr

Web: http://mobilelab.khu.ac.kr/