Object Oriented Programming in C++

In this Week

- Object Oriented Programming (OOP) Paradigm
- C++ Classes and Objects
- Member Variables and Member Functions
- Access Privileges: Private and Public
- Getters, Setters and Constructors
- The Dot and Scope Resolution operators
- Pointers and References to Objects
- Non-member functions
- Arrays of Objects

Classes in C++

- In our previous section, we discussed the C++ **struct** and the concept of data encapsulation in C++ structures
- More importantly, we noted how problem solving becomes more manageable using C++ structures
- In this section, we extend the idea further to encapsulate not only data but also functions!

Classes in C++

- In C++, **a class is a data type**
- Variables of a Class data type are called Objects
- A **class encapsulates both member variables** and **member functions** of objects designed to work with the member variables
- Object Oriented Programming (OOP) is a programming paradigm or methodology dealing with the design, implementation and maintenance of classes and objects and [main] programs that make use of them

Classes in C++

- A properly designed OOP identifies smallest objects and then builds bigger objects by combining these smaller objects
- For example, a **Point** in 2D space is an object that has x and y coordinates
- Similarly, a **Line** is an object that has two end **Points**
- Therefore, a **Line** object is made up two **Point** objects

C++ Class Syntax

- The complete syntax of C++ class is

```
class className
{
      public:
            member variables come here...
            member functions come here...
};
```

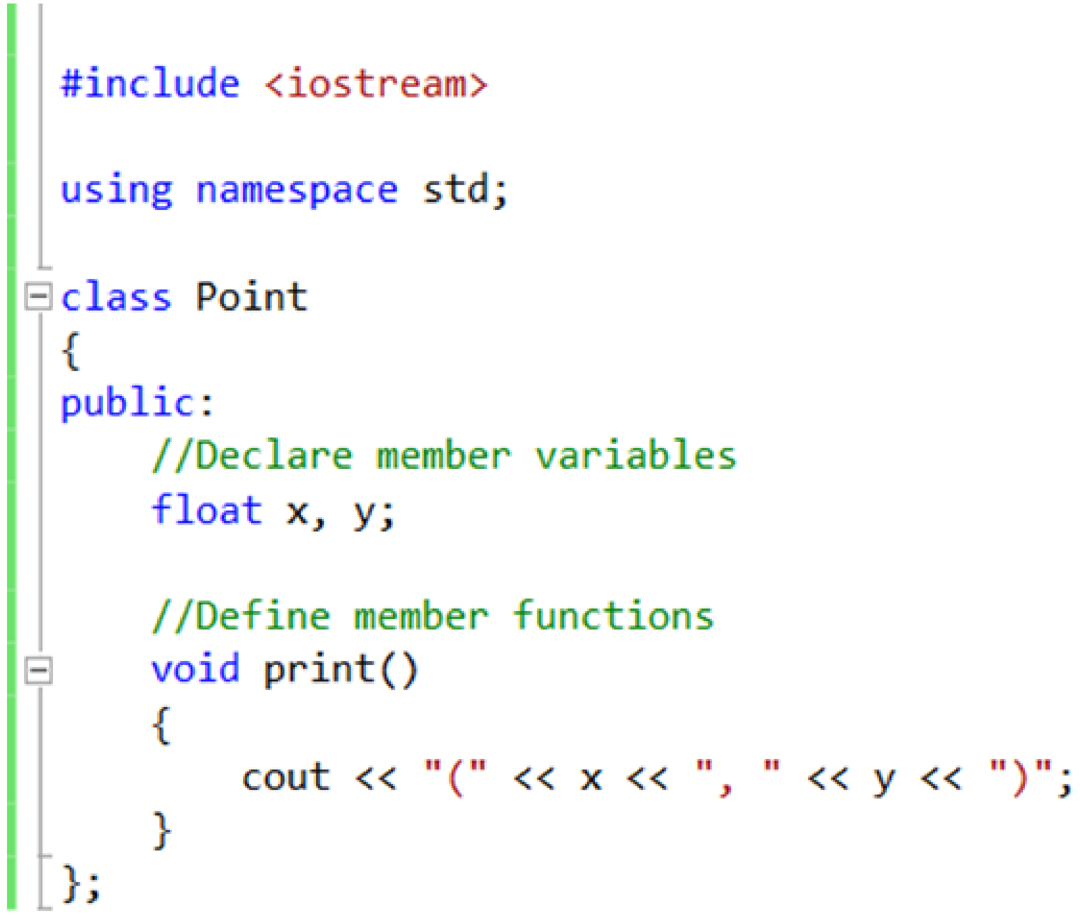
- A class starts with the keyword **class**
- The class name can be any string that follows the rules of variable naming
- The keyword **public** will be discussed later

C++ Class Example

- Let us design a simple class to demonstrate the idea of OOP
- For that, let us consider the **Point** class representing points in 2D space
- As member variable, a **Point** in 2D space has an **x-coordinate** and **y-coordinate**
- As member functions, a **Point** has a function to **print the Point object**

C++ Point Class Example

- Therefore the Point class may look like



```
#include <iostream>

using namespace std;

class Point
{
public:
	//Declare member variables
	float x, y;

	//Define member functions
	void print()
	{
		cout << "(" << x << ", " << y << ")";
	}
};
```

C++ Point Class Example

- The Point class is shown with minimal code
- It has two member variables: x and y coordinates and only one member function to print it
- A class can have as many member variables and member functions
- Now, we can write a main program and declare a variable (**object**) of type Point

C++ Point Class Example

- Just like in the C++ structures, we create objects by declaring a variable of class type as follows

  Point p;

- Objects use the dot operator to access their member variables and member functions
- Therefore, given a point object **p**,

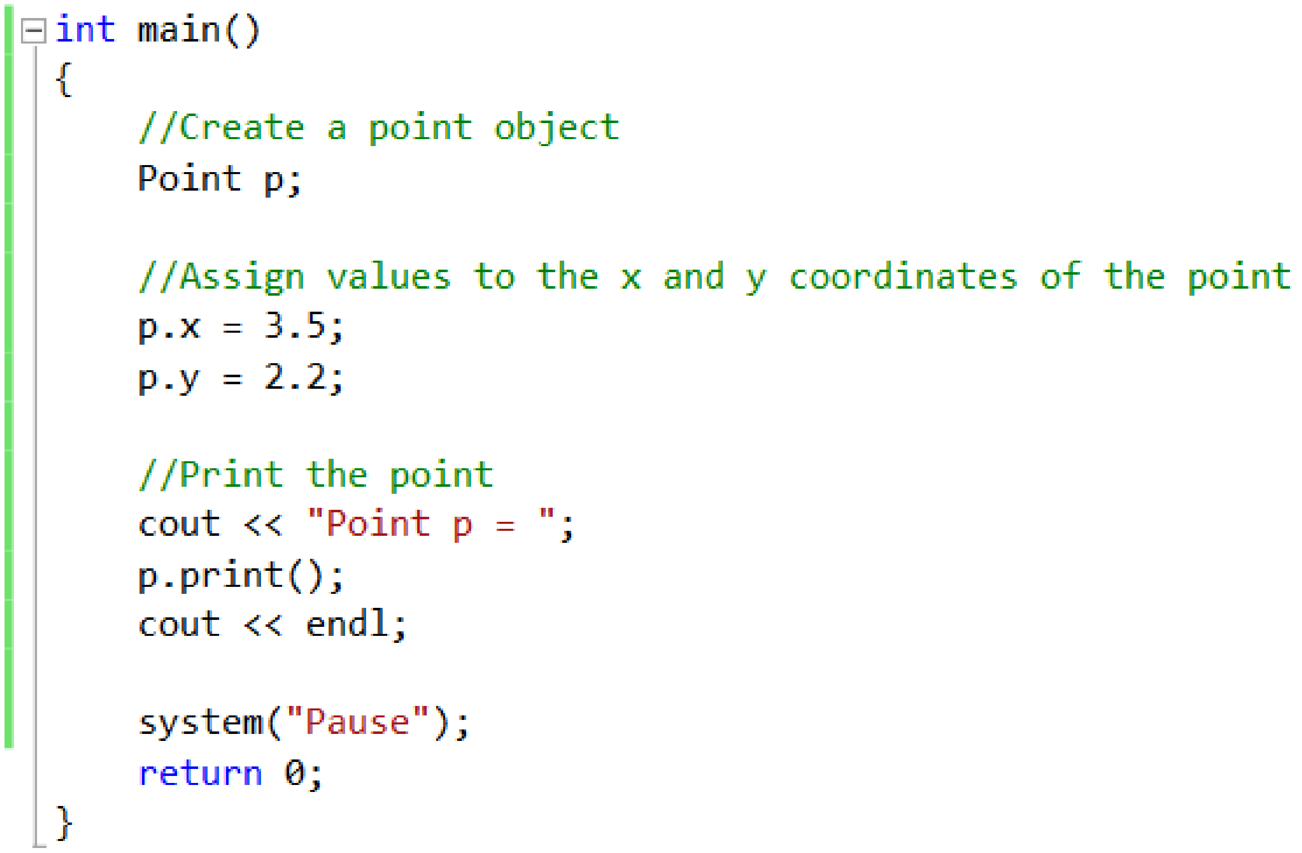
  p.x → accesses the x coordinate of the point object p. Similarly,

  p.print() → accesses the print function of the point object p

- Hence **p.x = 3.5;** assigns the literal value 3.5 to the x coordinate of the point object. Similarly for **p.y = 2.2;**

C++ Point Class Example

- Main program that makes use of the **Point** class



```
int main()
{
    //Create a point object
    Point p;

    //Assign values to the x and y coordinates of the point
    p.x = 3.5;
    p.y = 2.2;

    //Print the point
    cout << "Point p = ";
    p.print();
    cout << endl;

    system("Pause");
    return 0;
}
```

Design of Classes

- It is important classes are designed with care in mind so that to minimize the debugging and maintenance time of programs
- In the Point class example, the **main program has unrestricted access** to the member variables and member functions of Point class
- Is this good thing? **NO!**
- Imagine you have many functions and a main program that make use of the point object p. Now, if you modify the x-coordinate of p in one function but forgot that it will affect any subsequent usage of object p, then the subsequent computations will unintentionally use the modified x-coordinate value and you will get unexpected results
- It turns out, such errors are hard to find and the developers of C++ have provided a way to make sure no object or non-class member function directly modifies its member variables!

C++ Access Privileges: Public

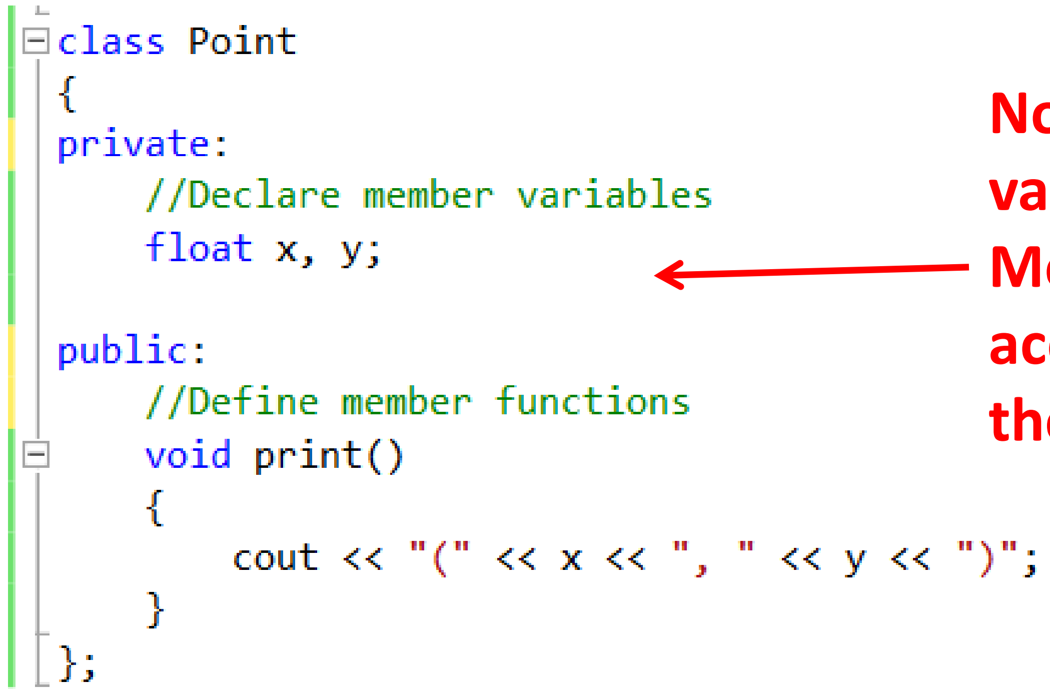
- As shown in the Point class example, there is the keyword **public**
- What does it mean?
- It means everything below it which is to say all the member variables and member functions are directly accessible to any Point object and any non-member functions (such as main fun.)
- This is known as **public access privilege!**

C++ Access Privileges: Private

- As described above, giving public privileges to member variables is a very dangerous design
- **Instead it is important to design a class such that the member variables are private while the member functions are public**
- **Private members are accessible ONLY inside the class!**
- This way, every communication to the member variables will be done through **member functions**

Point Class Version 2

- Now, consider the redesigned Point class (version 2) shown below



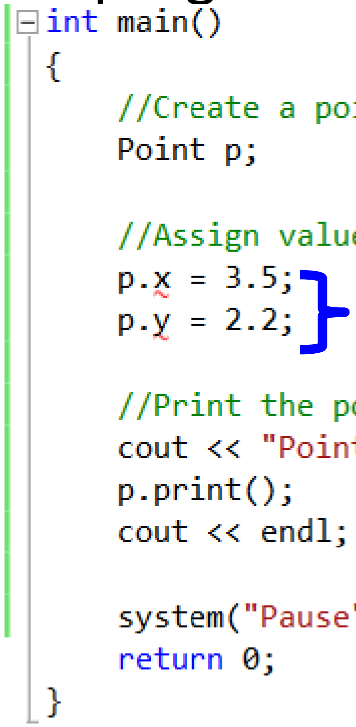
```
class Point
{
private:
    //Declare member variables
    float x, y;

public:
    //Define member functions
    void print()
    {
        cout << "(" << x << ", " << y << ")";
    }
};
```

Now both the member variables are private. Meaning they can be accessed only inside the class

- Will the main program still work? Let us see...

Point Class Version 2

- The main program is shown below



```
int main()
{
    //Create a point object
    Point p;

    //Assign values to the x and y coordinates of the point
    p.x = 3.5;
    p.y = 2.2;

    //Print the point
    cout << "Point p = ";
    p.print();
    cout << endl;

    system("Pause");
    return 0;
}
```

Incorrect statements. Object p can not access member variable x or y

- As you can see, the statements **p.x = 3.5;** and **p.y = 2.2;** are now **syntactically** incorrect! Why? Because the member variables x and y are not accessible from outside the class!

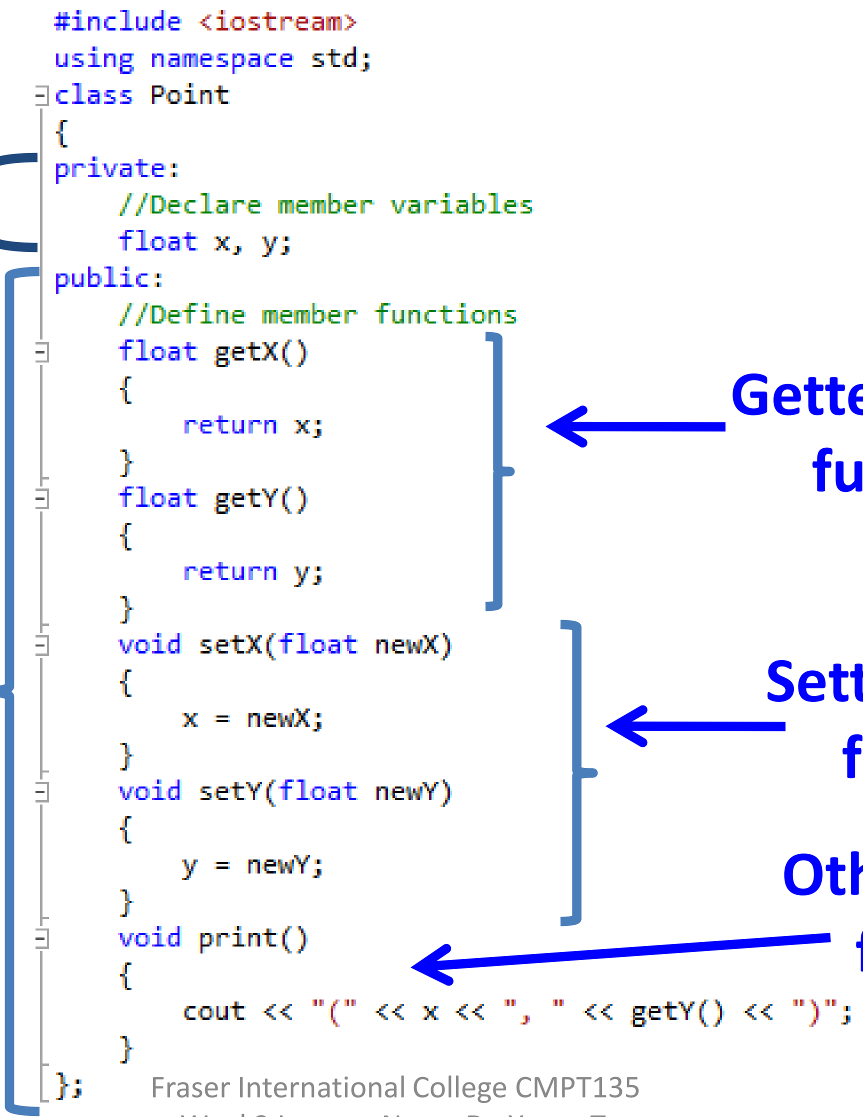
Point Class Version 3

- From the previous discussion, it is imperative that we provide public member functions through which we can access the private member variables
- Therefore in order to have a complete class, we provide functions to get the member variables and functions to set (assign values to) the member variables
- Such functions are commonly known as **getters** and **setters**! Getters return the member variables and setters assign them values
- The following Point class version demonstrates this...

Point Class Version 3

Private Member Variables

Public Member Functions



```cpp
#include <iostream>
using namespace std;
class Point
{
private:
    //Declare member variables
    float x, y;
public:
    //Define member functions
    float getX()
    {
        return x;
    }
    float getY()
    {
        return y;
    }
    void setX(float newX)
    {
        x = newX;
    }
    void setY(float newY)
    {
        y = newY;
    }
    void print()
    {
        cout << "(" << x << ", " << getY() << ")";
    }
};
```

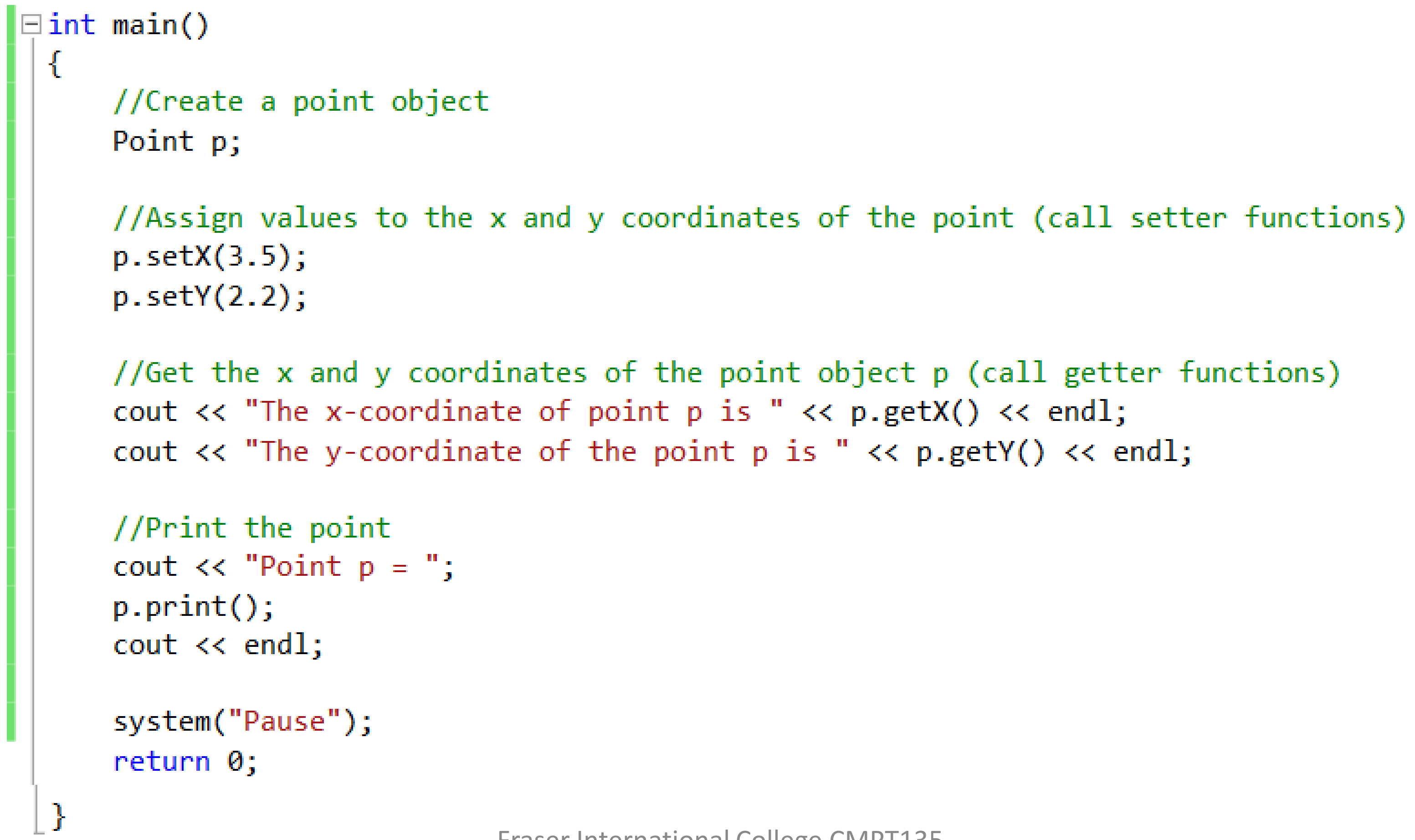
Getter member functions

Setter member functions

Other member functions

Using Point Class in Main Program

- The main program that makes use of the point class is shown below...



```cpp
int main()
{
    //Create a point object
    Point p;

    //Assign values to the x and y coordinates of the point (call setter functions)
    p.setX(3.5);
    p.setY(2.2);

    //Get the x and y coordinates of the point object p (call getter functions)
    cout << "The x-coordinate of point p is " << p.getX() << endl;
    cout << "The y-coordinate of the point p is " << p.getY() << endl;

    //Print the point
    cout << "Point p = ";
    p.print();
    cout << endl;

    system("Pause");
    return 0;
}
```

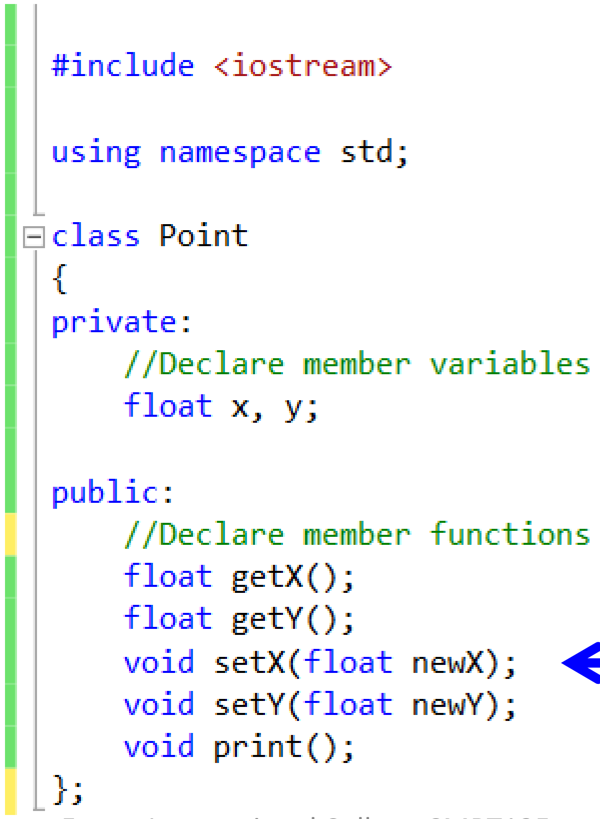
Member Functions Outside Class

- Look at the Point class version 3 shown above
- As the number of member variables and member functions become more, it is obvious the class will grow substantially and it might be hard to see clearly the functionality of the class
- Thus, it is possible to only put the **declarations** of the **member variables** and **member functions** inside the class and put the implementation of the member functions outside the class as shown next...

Point Class Version 4

- In version 4, we present the working version 3 Point class but this time, we give only declarations inside the class...

Neat presentation of a class



```
#include <iostream>

using namespace std;

class Point
{
private:
	//Declare member variables
	float x, y;

public:
	//Declare member functions
	float getX();
	float getY();
	void setX(float newX);
	void setY(float newY);
	void print();
};
```

Function declarations have semicolons

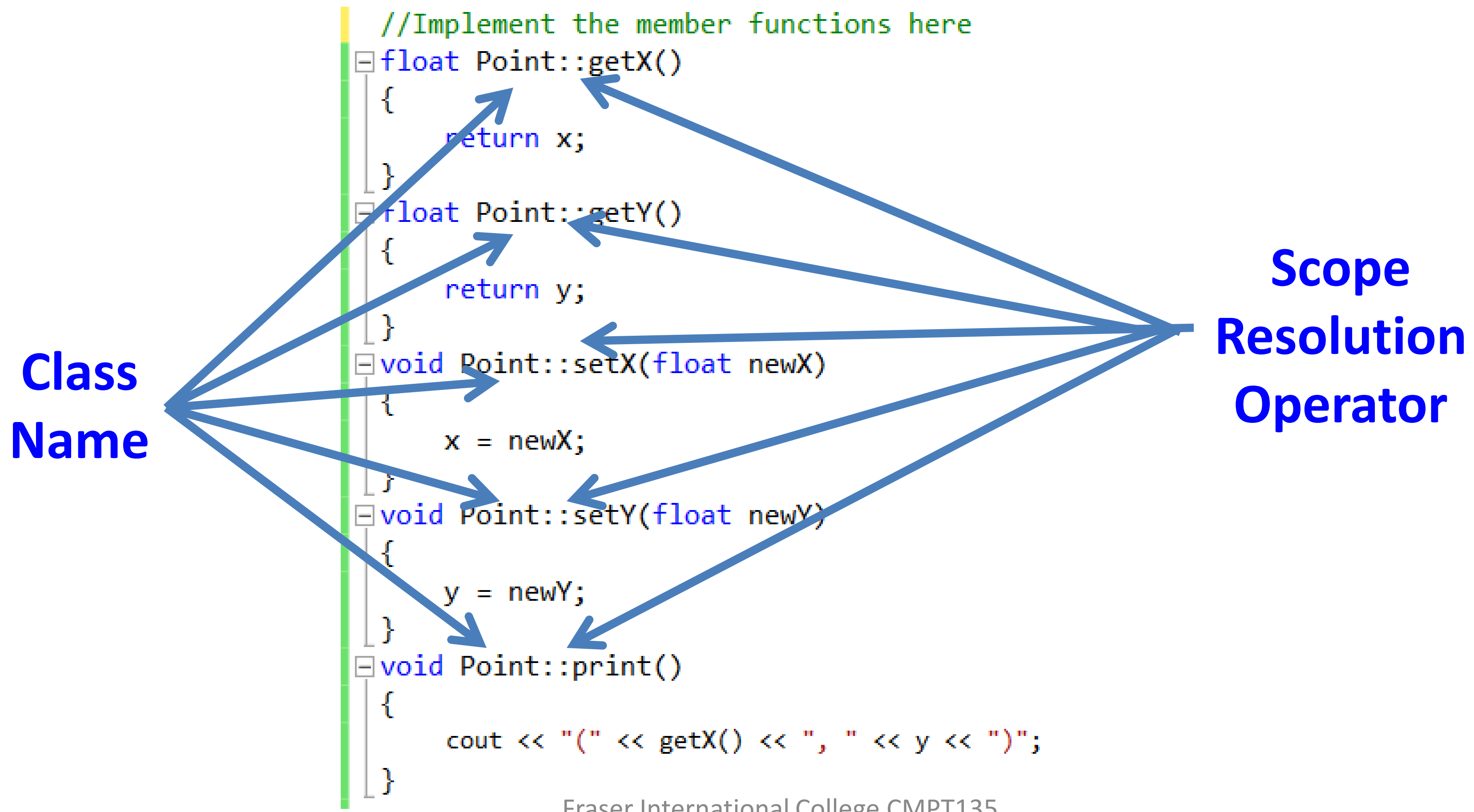
Member Functions Outside Class

- Now, we attempt to implement the member functions just below the class and above the main program
- Of course, we can even put the implementation below the main program if we wish
- The question is how do we tell C++ a certain function, say getX() belongs to the Point class?
- **Answer**: **We use the scope resolution operator ::**
- The function then will look like

```
float Point::getX()
{
    return x;
}
```

Member Functions Outside Class

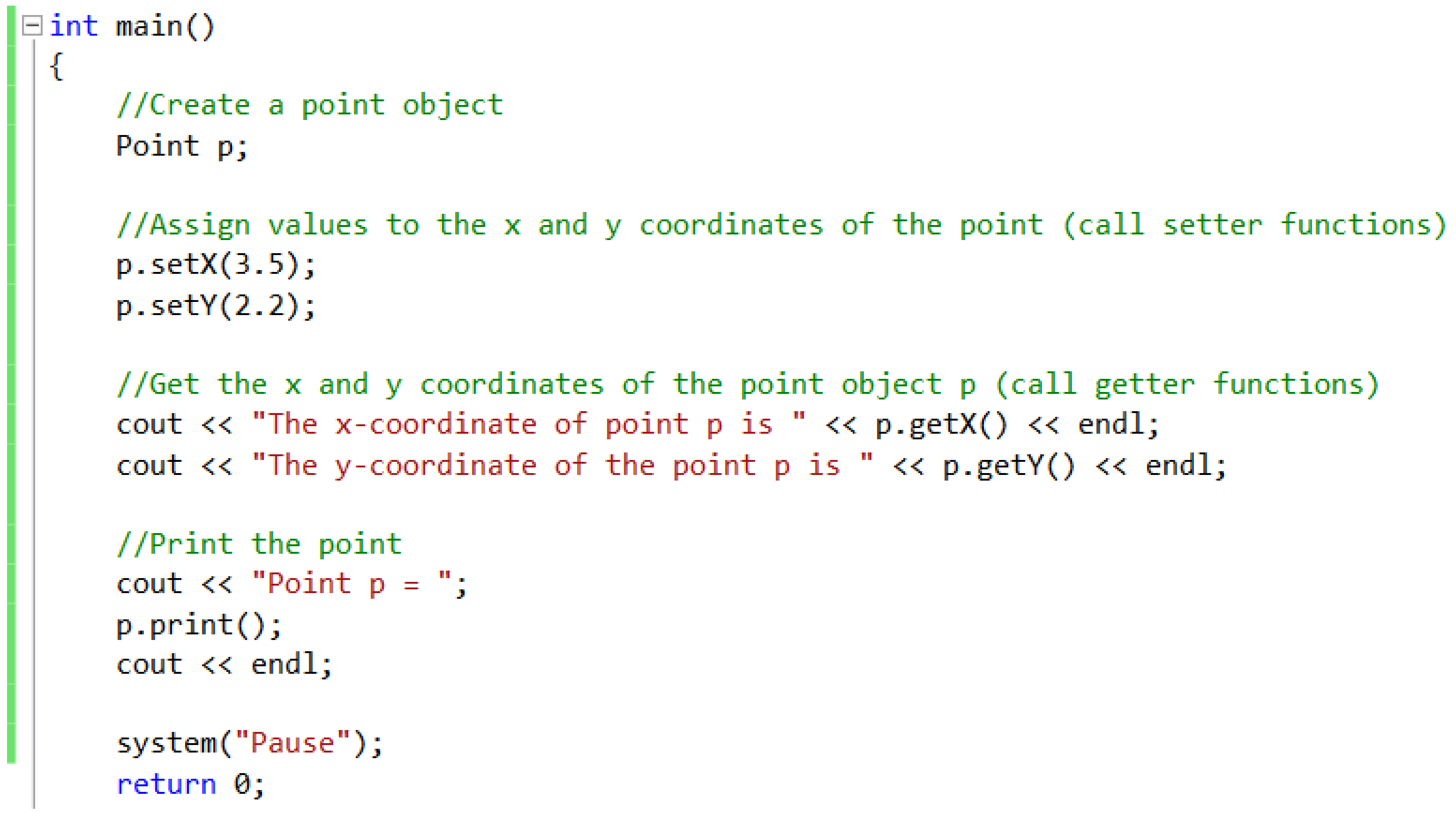
- Below is the implementation of the member functions



```
//Implement the member functions here
float Point::getX()
{
        return x;
}
float Point::getY()
{
        return y;
}
void Point::setX(float newX)
{
        x = newX;
}
void Point::setY(float newY)
{
        y = newY;
}
void Point::print()
{
        cout << "(" << getX() << ", " << y << ")";
}
```

Member Functions Outside Class

- Main program...



```
int main()
{
    //Create a point object
    Point p;

    //Assign values to the x and y coordinates of the point (call setter functions)
    p.setX(3.5);
    p.setY(2.2);

    //Get the x and y coordinates of the point object p (call getter functions)
    cout << "The x-coordinate of point p is " << p.getX() << endl;
    cout << "The y-coordinate of the point p is " << p.getY() << endl;

    //Print the point
    cout << "Point p = ";
    p.print();
    cout << endl;

    system("Pause");
    return 0;
```

The Assignment Operator and Objects

- Assignment Operator
  - ➢ Just like with simple data types, we can assign an object to another object of the same type
  - ➢ The assignment will be performed in one to one correspondence of the member variables
- Thus given two Point objects p1 and p2 and assuming p1 has already been initialized, the statement

  ```
  p2 = p1;
  ```

  Will assign the value of the x member variable of p1 to the x member variable of p2; and the value of the y member variable of p1 to the y member variable of p2

Constructor Member Functions

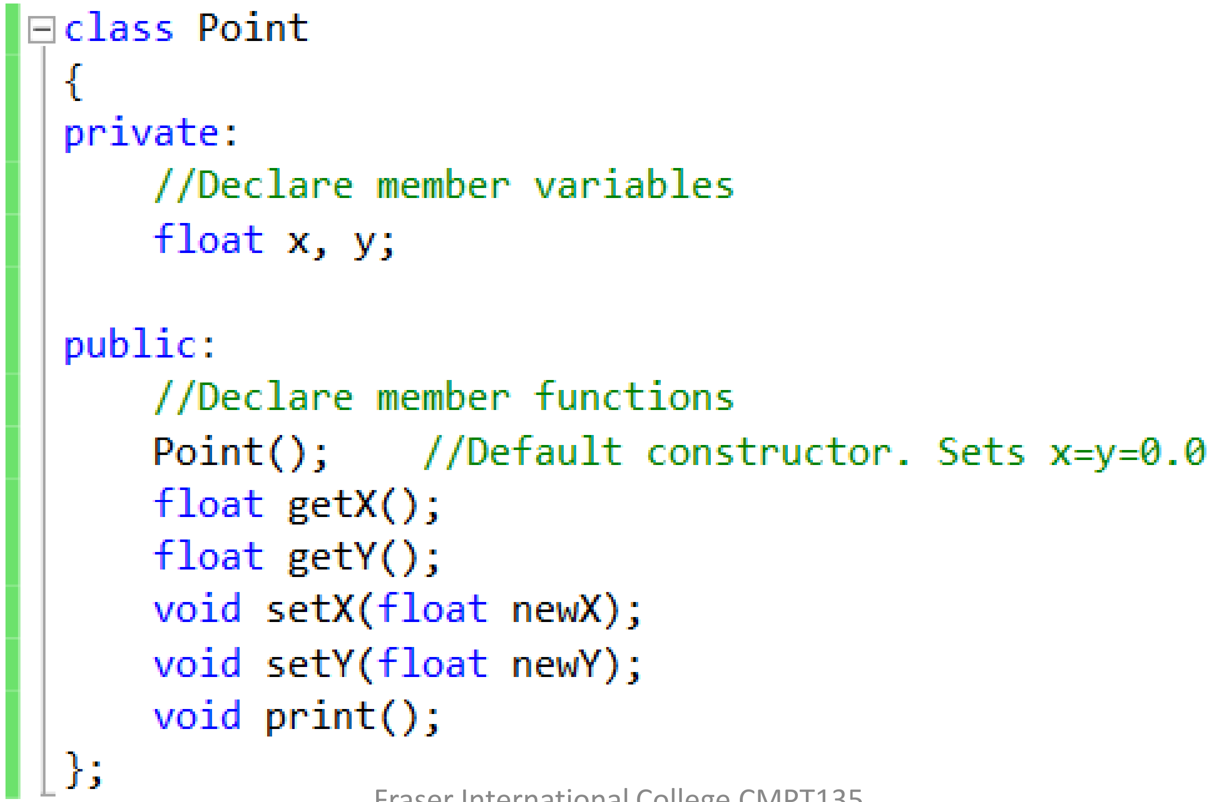
- Consider the statement

  Point p;

  in the main program shown earlier
- What happens if we try to print the Point object p right after declaration BUT before initialization?
- **Answer**: Some garbage will be printed because the x and y coordinates have not been set yet
- Sometimes, however we may wish to have some default point to be created after declaration; say the origin
- In order to have such capability, we need to add a special member function called Constructor Member Function that initializes default values for the member variables
- Constructors MUST have the same name as the class name. Moreover constructors can NOT return any value; not even void!!!

Constructor Member Functions

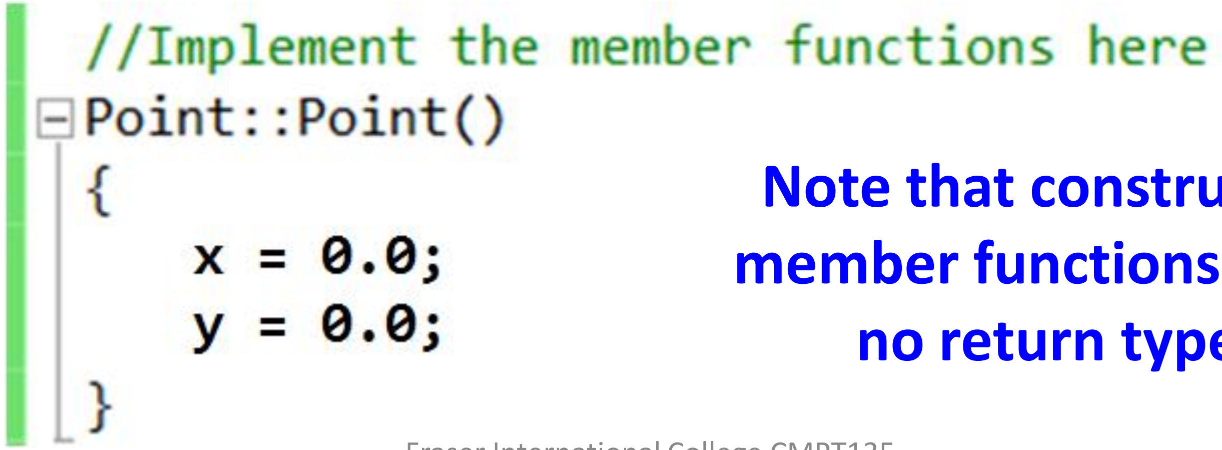
- The Point class with constructor member function is shown below



```
class Point
{
private:
    //Declare member variables
    float x, y;

public:
    //Declare member functions
    Point();    //Default constructor. Sets x=y=0.0
    float getX();
    float getY();
    void setX(float newX);
    void setY(float newY);
    void print();
};
```

Constructor Member Functions

- The implementation of the default constructor member function is the same as any other member function except that constructor does not return anything NOT EVEN VOID
- Therefore it looks like:



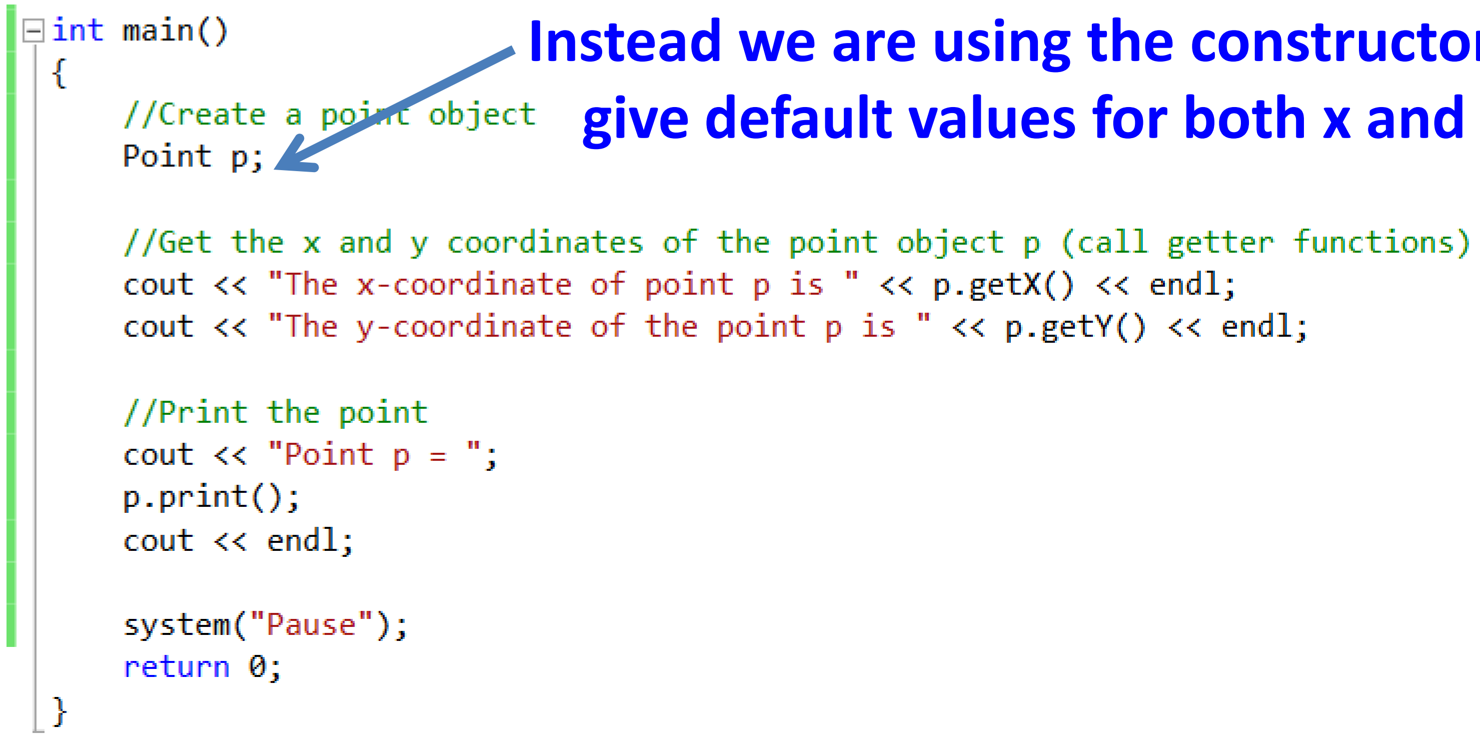
```
//Implement the member functions here
Point::Point()
{
    x = 0.0;
    y = 0.0;
}
```

Note that constructor member functions have no return type

Constructor Member Functions

- Now the following main program will create a point p that is at the origin and therefore printing it will give 0.0 for both x and y

We have not called the setter functions. Instead we are using the constructor to give default values for both x and y



```
int main()
{
    //Create a point object
    Point p;

    //Get the x and y coordinates of the point object p (call getter functions)
    cout << "The x-coordinate of point p is " << p.getX() << endl;
    cout << "The y-coordinate of the point p is " << p.getY() << endl;

    //Print the point
    cout << "Point p = ";
    p.print();
    cout << endl;

    system("Pause");
    return 0;
}
```

Non-default Constructors

- Observe that any point object created now will by default be the origin unless we call its setters
- BUT, what if we would like to create a point of our choice at the same time as we declare a point object
- Suppose we would like to create a point with x coordinate equal to 1.0 and y coordinate equal to 2.0
- Wouldn't it be nice to have something like

```
Point p(1.0, 2.0);
```

- This way we could initialize the point object at the time of declaration with non-default values...

Non-default Constructors

- In order to be able to do so, C++ provides the ability to have constructors that take arguments
- This way the incoming argument values are assigned to the member variables
- It is customary to call setter functions whenever we assign values to member variables; therefore constructors typically call setters in order to assign values...

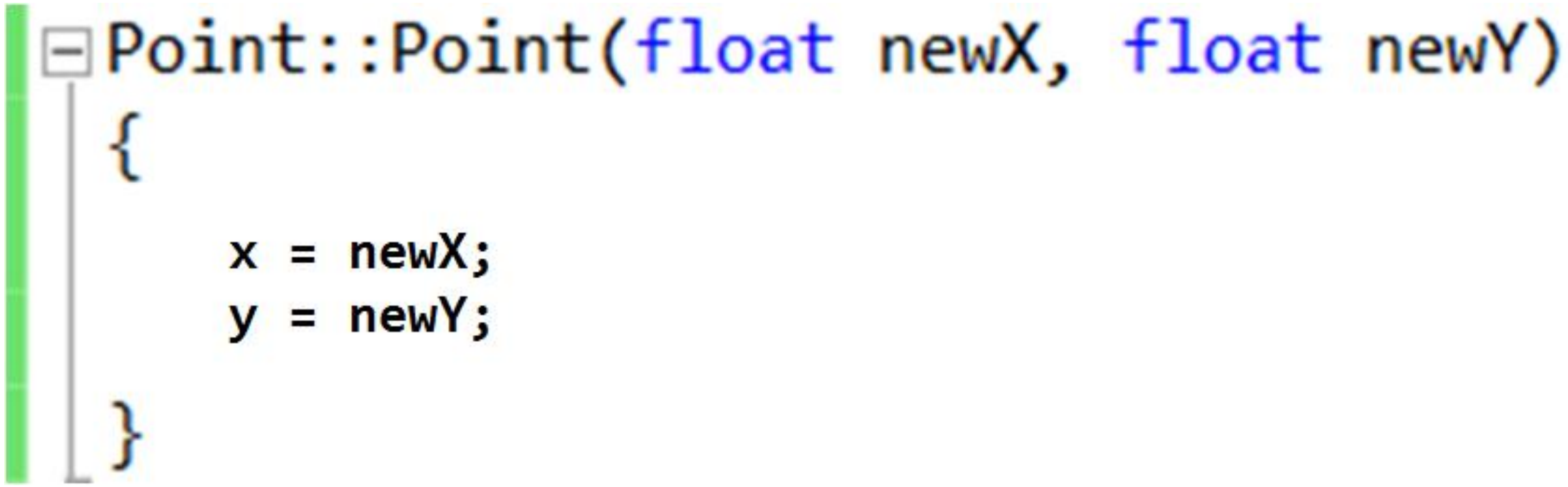
Non-default Constructors

- The Point class with the non-default constructor added to it will look like

```
#include <iostream>

using namespace std;

class Point
{
private:
    //Declare member variables
    float x, y;

public:
    //Declare member functions
    Point();    //Default constructor. Sets x=y=0.0
    Point(float newX, float newY);  //non-default constructor
    float getX();
    float getY();
    void setX(float newX);
    void setY(float newY);
    void print();
};
```

Non-default Constructors

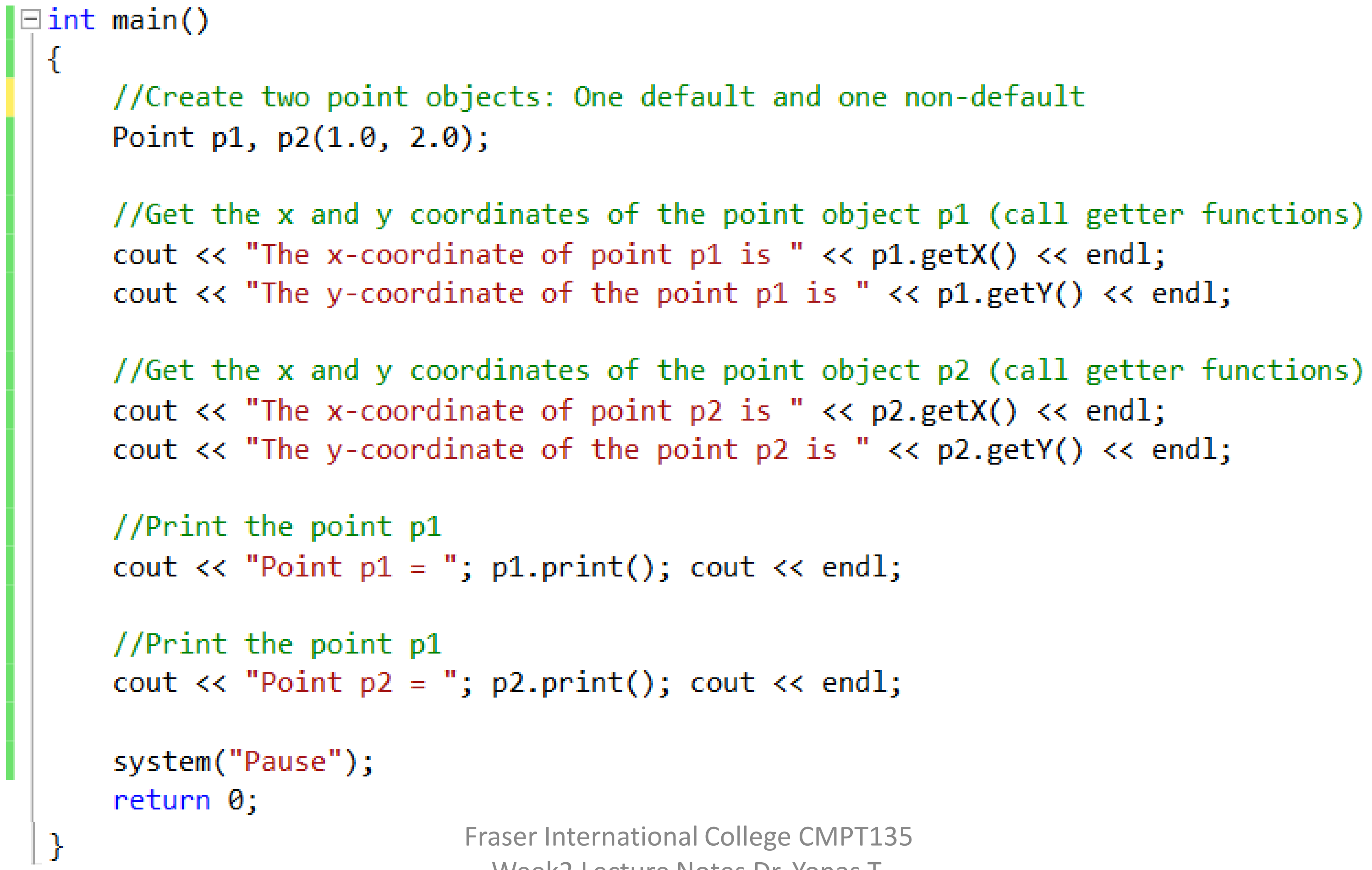
- The implementation of the non-default constructor follows same procedure



```
Point::Point(float newX, float newY)
{

    x = newX;
    y = newY;

}
```

Using Default and Non-default Constructors

- The following main program demonstrates the use of constructors...



```
int main()
{
    //Create two point objects: One default and one non-default
    Point p1, p2(1.0, 2.0);

    //Get the x and y coordinates of the point object p1 (call getter functions)
    cout << "The x-coordinate of point p1 is " << p1.getX() << endl;
    cout << "The y-coordinate of the point p1 is " << p1.getY() << endl;

    //Get the x and y coordinates of the point object p2 (call getter functions)
    cout << "The x-coordinate of point p2 is " << p2.getX() << endl;
    cout << "The y-coordinate of the point p2 is " << p2.getY() << endl;

    //Print the point p1
    cout << "Point p1 = "; p1.print(); cout << endl;

    //Print the point p1
    cout << "Point p2 = "; p2.print(); cout << endl;

    system("Pause");
    return 0;
}
```

Some Remarks on Constructors

- Constructors can ONLY be called at the time of declaration. We can not declare an object and then call its constructor.
- Therefore

  Point p;

  p.Point(); ← **Incorrect**

  p.Point(1.0, 2.5); ← **Incorrect**

  are incorrect syntaxes; because constructors can not be called as ordinary member functions. Instead, what we should do is

  Point p(1.0, 2.5); ← **Correct**

- Alternatively, we may create default object and then replace it as follows

  Point p;

  p = Point(1.0, 2.5); ← **Correct**

- Alternatively, call setter functions to change values of x or y coordinates.

Some Remarks on Constructors

- Observe that the statement

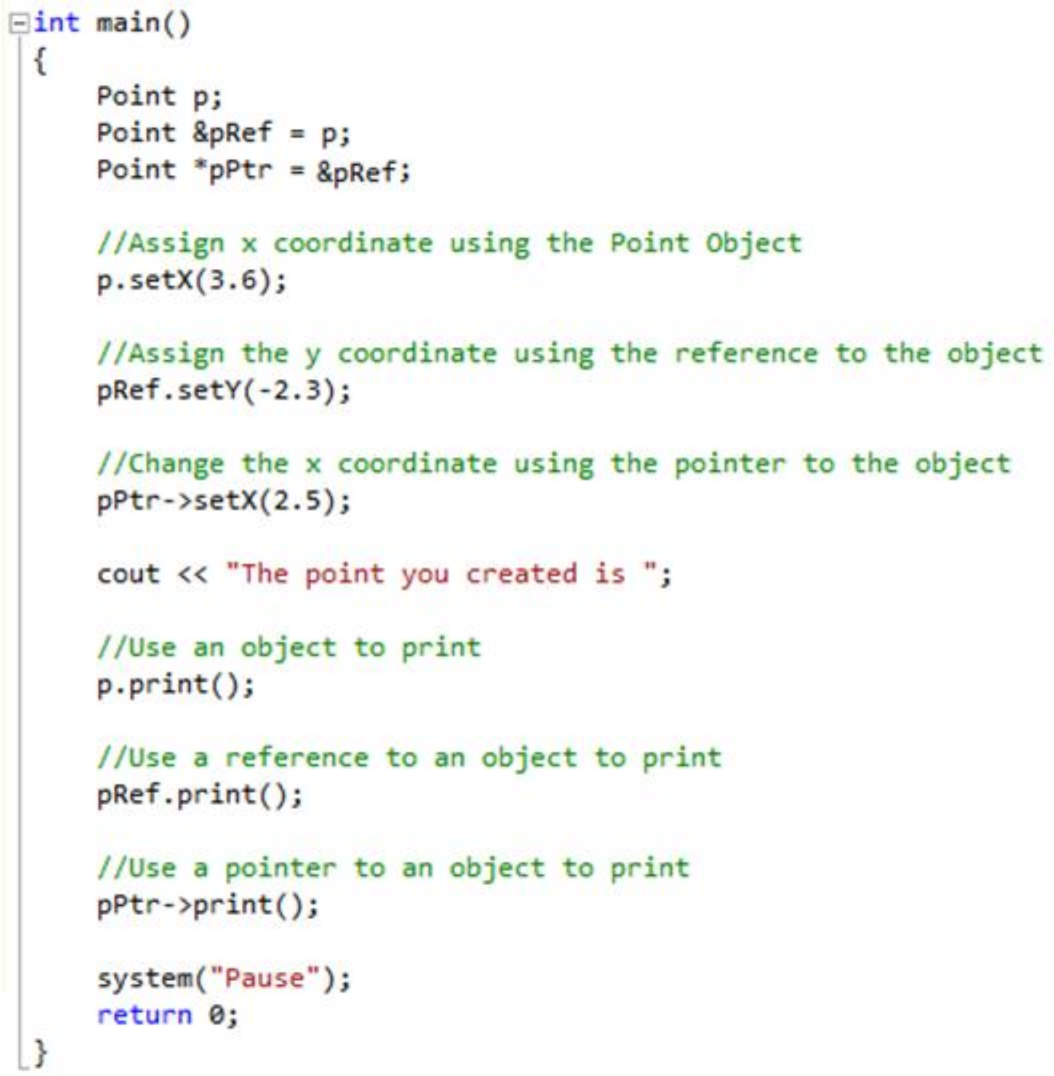
  Point p;

- This calls the default constructor
- But what if there is no default constructor in our class; will the statement still be valid?
- **Answer:- YES! C++ provides a default constructor whenever our class does NOT have a default constructor**
- The C++ provided default constructor has an empty body; thus the x and y member variables will not be initialized by it
- Whenever we provide default constructor, it replaces the C++ provided default constructor
- Moreover **whenever we define non-default constructor, then we MUST also define a default constructor**

Pointers and References to Objects

- Just like any other variable, we can declare pointers to objects
- Pointers can be assigned either the memory address of an actual object or we may create a new object pointed by a pointer using the new operator
- In order to dereference and access member variables and member functions of pointers to objects, we use the **->** operator
- The **->** operator is equivalent to dereferencing and the using the dot operator. Thus given a pointer **pPtr**, the statement **pPtr->print()** is equivalent to **(*pPtr).print()**

Pointers and References to Objects



```
int main()
{
    Point p;
    Point &pRef = p;
    Point *pPtr = &pRef;

    //Assign x coordinate using the Point Object
    p.setX(3.6);

    //Assign the y coordinate using the reference to the object
    pRef.setY(-2.3);

    //Change the x coordinate using the pointer to the object
    pPtr->setX(2.5);

    cout << "The point you created is ";

    //Use an object to print
    p.print();

    //Use a reference to an object to print
    pRef.print();

    //Use a pointer to an object to print
    pPtr->print();

    system("Pause");
    return 0;
}
```

Constructing Objects on the Heap Memory

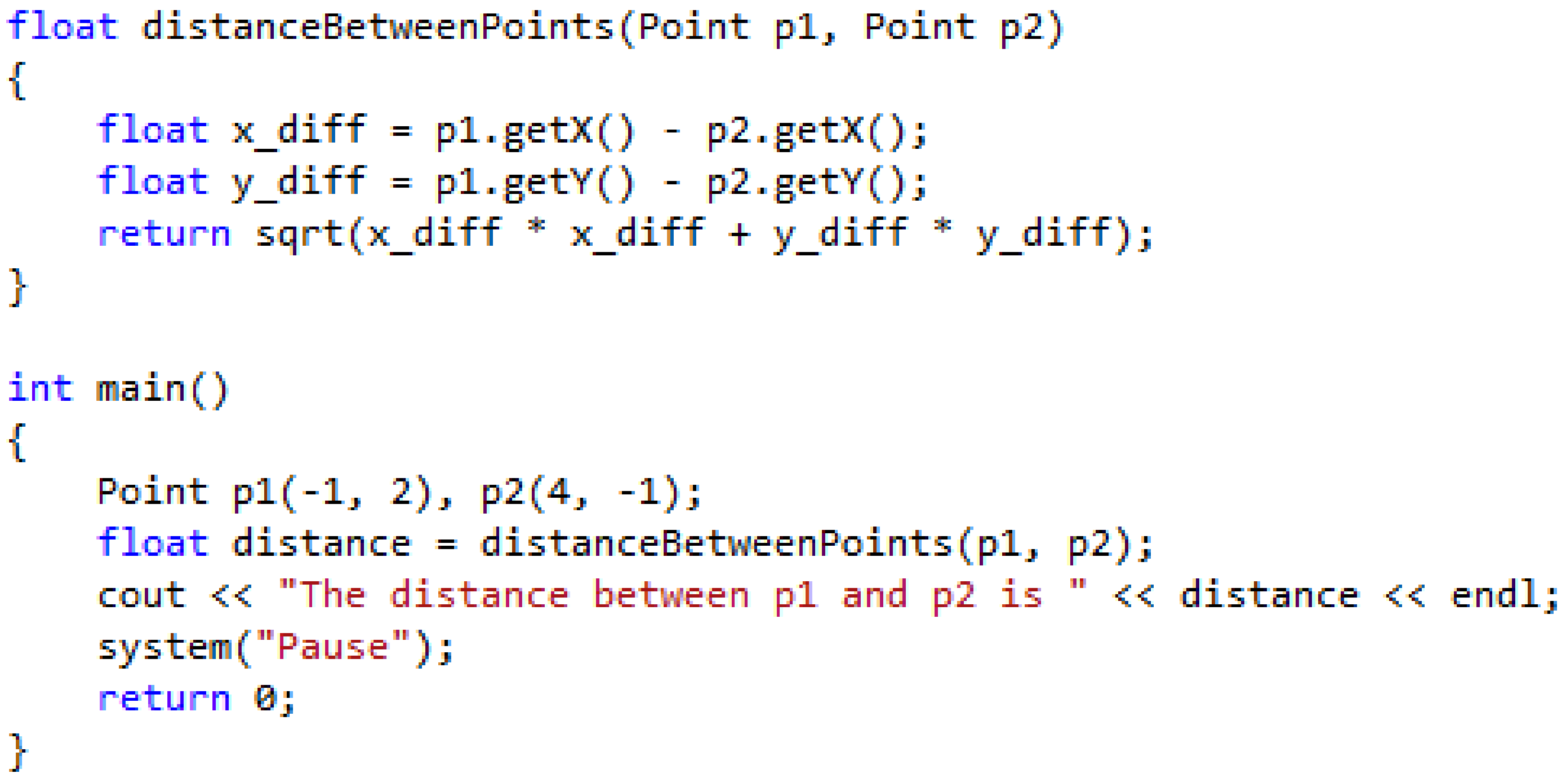
- Similarly we can use pointers to create objects on the heap using the new operator
- When the objects on the heap memory are no more needed; we should free their memory using the delete operator. Analyze the following program and determine its output

```
int main()
{
    Point *p1, *p2, *p3;
    p1 = new Point;                    //Default object constructed
    p2 = new Point();                  //Default object constructed
    p3 = new Point(2.5, -1.8);         //Non-default object constructed
    Point *p4 = new Point(2.2, 5.3);   //Non-default object constructed

    cout << "Point p1 is "; p1->print(); cout << endl;
    cout << "Point p2 is "; p2->print(); cout << endl;
    cout << "Point p3 is "; p3->print(); cout << endl;
    cout << "Point p4 is "; p4->print(); cout << endl;

    delete p1;
    delete p2;
    delete p3;
    delete p4;

    system("Pause");
    return 0;
}
```

Non-Member Functions

- A given C++ program may also have non-member functions in addition to class declaration, member functions definitions and main program
- A non-member function is simply a C++ function which does not belong to a class but rather is designed to work with objects of a class type for a particular computation
- We may use parameter passing by value, by reference or by pointer
- Moreover a non-member function may return any data type including a class type
- For example, we may write a non-member function named **`distanceBetweenPoints`** that takes two Point objects and returns the distance between them as follows

Non-Member Functions



```
float distanceBetweenPoints(Point p1, Point p2)
{
    float x_diff = p1.getX() - p2.getX();
    float y_diff = p1.getY() - p2.getY();
    return sqrt(x_diff * x_diff + y_diff * y_diff);
}

int main()
{
    Point p1(-1, 2), p2(4, -1);
    float distance = distanceBetweenPoints(p1, p2);
    cout << "The distance between p1 and p2 is " << distance << endl;
    system("Pause");
    return 0;
}
```

- Please note that for reasons that will be described later, the parameters of the function must not have constant modifier

Arrays of Objects

- Just like with any other data types, we can also have an array (static or dynamic) of objects
- For example, we could declare a static array of ten Points and work with the array as follows

```
Point pointArray[10];
```

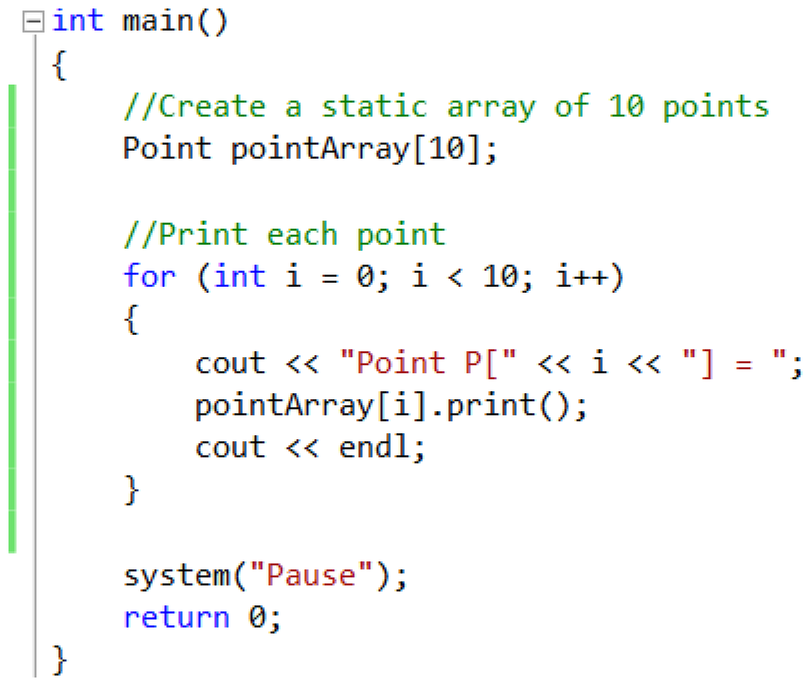
- Equally, we could have a dynamic array of size ten as follows

```
int size;
cin >> size;
Point *pointArray = new Point[size];
```

- In both cases, each element of the array will be Point object
- **Moreover the default constructor will be automatically invoked for each element and each of the arrays (the static or the dynamic) contains origin Points of ten elements**

Arrays of Objects

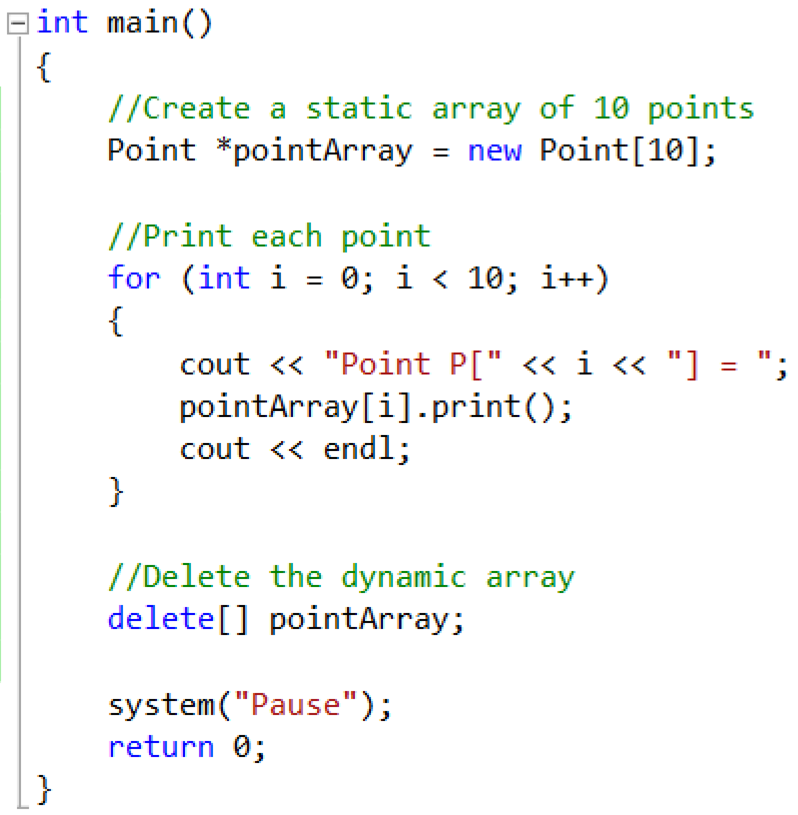
- The following code creates a static array of 10 points and print them
- The default constructor is automatically called for each element and therefore each point is the origin



```
int main()
{
    //Create a static array of 10 points
    Point pointArray[10];

    //Print each point
    for (int i = 0; i < 10; i++)
    {
        cout << "Point P[" << i << "] = ";
        pointArray[i].print();
        cout << endl;
    }

    system("Pause");
    return 0;
}
```

Arrays of Objects

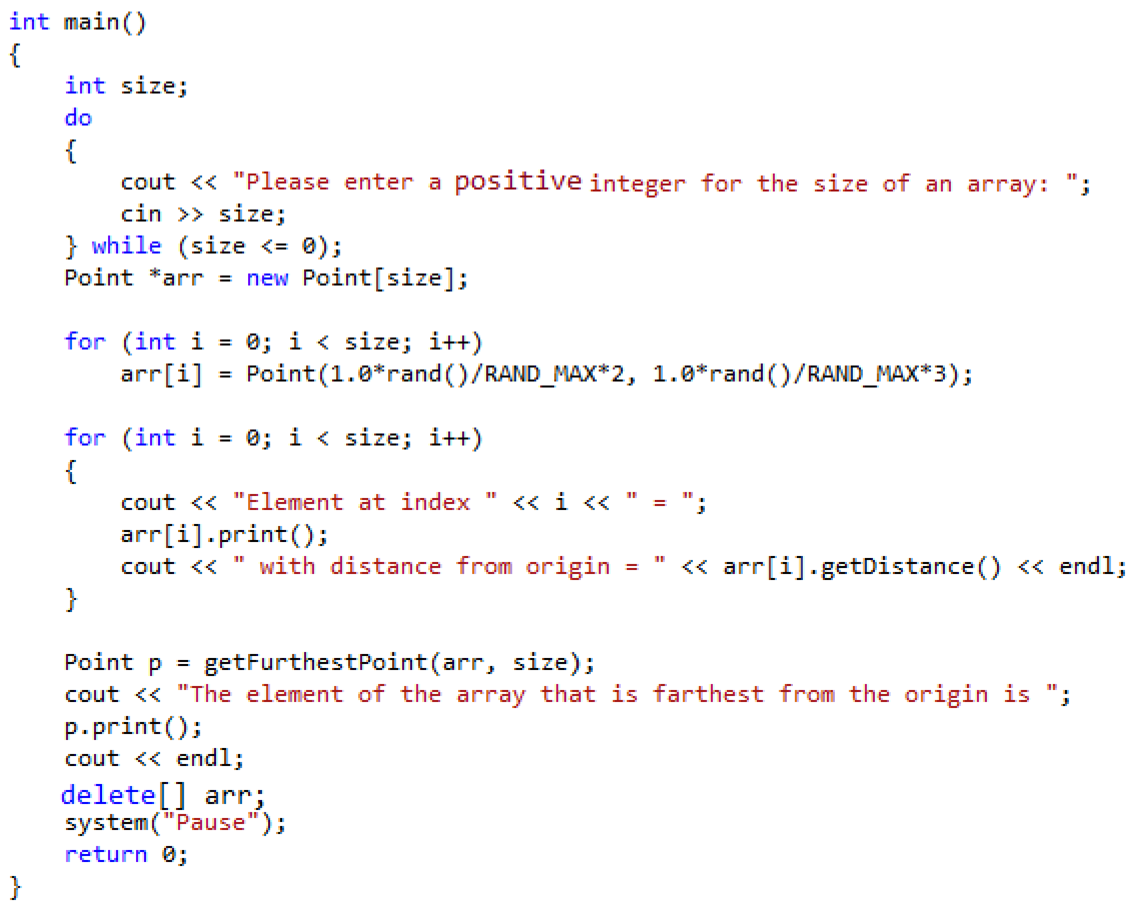
- The following code creates a dynamic array of 10 points and print them. Each element is a Point at the origin, why?



```
int main()
{
    //Create a static array of 10 points
    Point *pointArray = new Point[10];

    //Print each point
    for (int i = 0; i < 10; i++)
    {
        cout << "Point P[" << i << "] = ";
        pointArray[i].print();
        cout << endl;
    }

    //Delete the dynamic array
    delete[] pointArray;

    system("Pause");
    return 0;
}
```

Arrays of Objects

- Consider the following program and get it to work correctly by implementing the missing functions



```
int main()
{
    int size;
    do
    {
        cout << "Please enter a positive integer for the size of an array: ";
        cin >> size;
    } while (size <= 0);
    Point *arr = new Point[size];

    for (int i = 0; i < size; i++)
        arr[i] = Point(1.0*rand()/RAND_MAX*2, 1.0*rand()/RAND_MAX*3);

    for (int i = 0; i < size; i++)
    {
        cout << "Element at index " << i << " = ";
        arr[i].print();
        cout << " with distance from origin = " << arr[i].getDistance() << endl;
    }

    Point p = getFurthestPoint(arr, size);
    cout << "The element of the array that is farthest from the origin is ";
    p.print();
    cout << endl;
    delete[] arr;
    system("Pause");
    return 0;
}
```

Class As DataTypes of Member Variables of Other Classes

- Consider Line objects in 2D space
- Well a Line object is described by its two end points
- Therefore we may write a Line class which will have two member variables of Point data type where Point is a class on its own
- The declaration of Line class is shown below

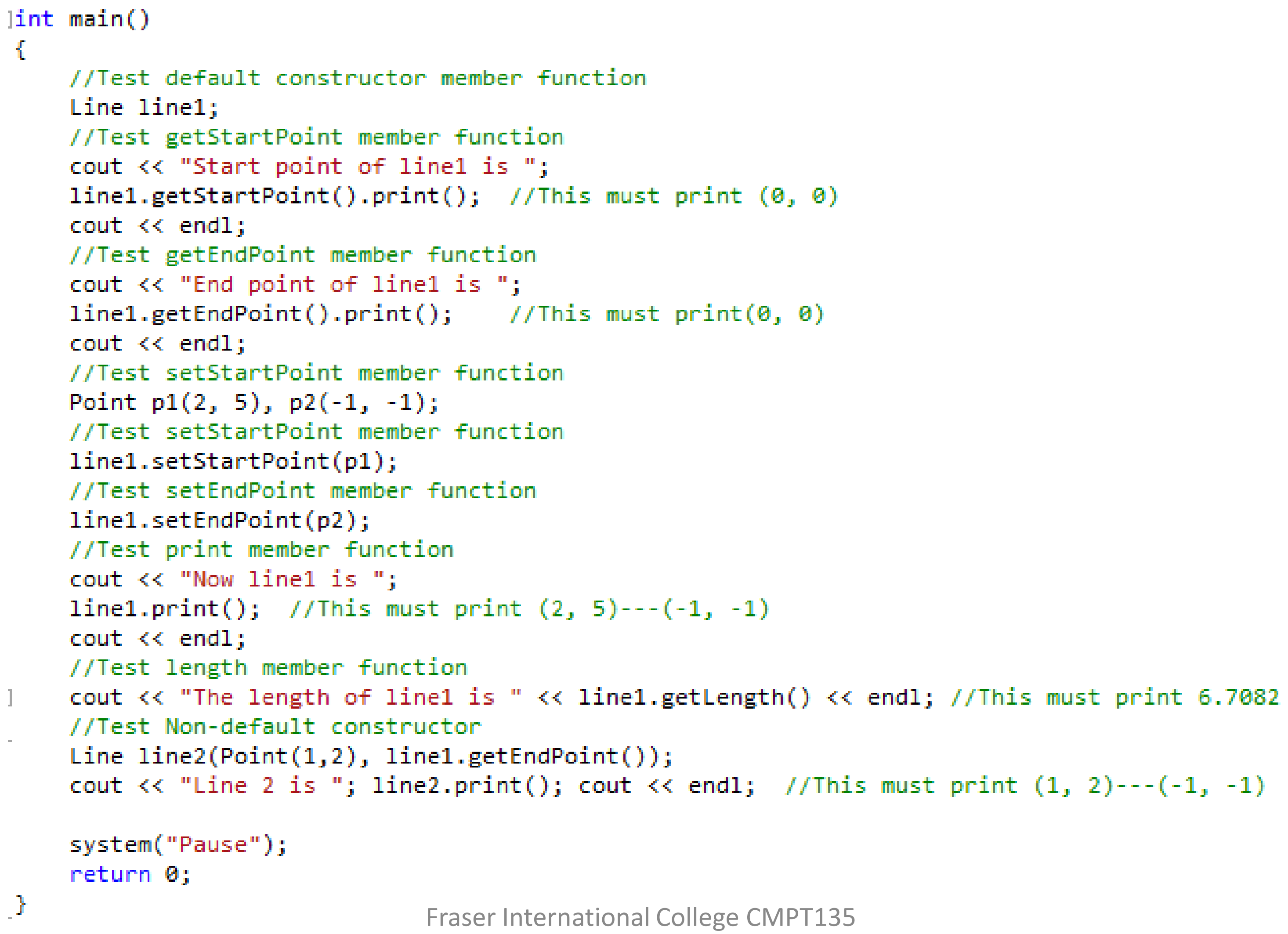
Class As DataTypes of Member Variables of Other Classes

```
class Line
{
private:
    Point start, end;
public:
    //Constructor member functions
    Line();//Make both start and end member variables default Point objects
    Line(Point s, Point e);

    //Getters
    Point getStartPoint();
    Point getEndPoint();

    //Setters
    void setStartPoint(Point s);
    void setEndPoint(Point e);

    //Additional member functions
    float getLength();
    void print();
};
```

Class As DataTypes of Member Variables of Other Classes

- Of course in order for this class to be valid, the declaration of the Point class must be placed above the Line class
- The implementations of the member functions of the Point class can come either before or after the declarations of the member functions of the Line class
- Exercise: Give the implementation of the Line class and test your class with the following test main program

Class As DataTypes of Member Variables of Other Classes

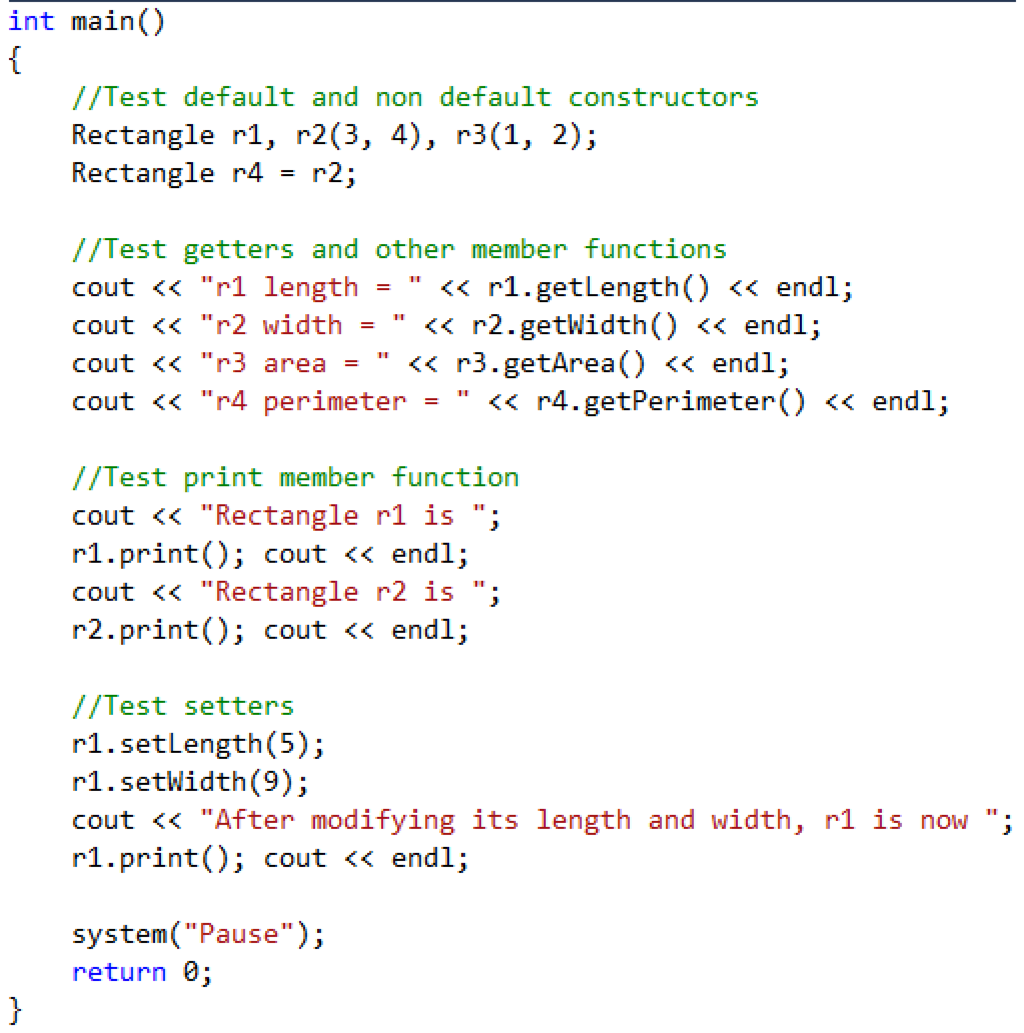


```
int main()
{
    //Test default constructor member function
    Line line1;
    //Test getStartPoint member function
    cout << "Start point of line1 is ";
    line1.getStartPoint().print();  //This must print (0, 0)
    cout << endl;
    //Test getEndPoint member function
    cout << "End point of line1 is ";
    line1.getEndPoint().print();    //This must print(0, 0)
    cout << endl;
    //Test setStartPoint member function
    Point p1(2, 5), p2(-1, -1);
    //Test setStartPoint member function
    line1.setStartPoint(p1);
    //Test setEndPoint member function
    line1.setEndPoint(p2);
    //Test print member function
    cout << "Now line1 is ";
    line1.print();  //This must print (2, 5)---(-1, -1)
    cout << endl;
    //Test length member function
    cout << "The length of line1 is " << line1.getLength() << endl; //This must print 6.7082
    //Test Non-default constructor
    Line line2(Point(1,2), line1.getEndPoint());
    cout << "Line 2 is "; line2.print(); cout << endl;  //This must print (1, 2)---(-1, -1)

    system("Pause");
    return 0;
}
```

Practice Example

- Write a C++ class named **Rectangle** that represents rectangle objects.
- A Rectangle should have two double data type member variables preferably named **length** and **width**
- Add a **default** constructor that assigns length and width member variables the default value of 1.0 each
- Add a **non default constructor** that takes two arguments and assigns the arguments to the member variables
- Add **getters** (getLength and getWidth) and **setters** (setLength and setWidth)
- Add a member function named **getArea** that does not take any argument and returns the area of the rectangle
- Add a member function named **getPerimeter** that does not take any argument and returns the perimeter of the rectangle given by 2*length + 2*width
- Add a member function named **print** that does not take any argument and prints the rectangle object appropriately (for example we may print the length, width, area, and perimeter in a nice format)
- Use the program given below to test your class design

Practice Example



```cpp
int main()
{
    //Test default and non default constructors
    Rectangle r1, r2(3, 4), r3(1, 2);
    Rectangle r4 = r2;

    //Test getters and other member functions
    cout << "r1 length = " << r1.getLength() << endl;
    cout << "r2 width = " << r2.getWidth() << endl;
    cout << "r3 area = " << r3.getArea() << endl;
    cout << "r4 perimeter = " << r4.getPerimeter() << endl;

    //Test print member function
    cout << "Rectangle r1 is ";
    r1.print(); cout << endl;
    cout << "Rectangle r2 is ";
    r2.print(); cout << endl;

    //Test setters
    r1.setLength(5);
    r1.setWidth(9);
    cout << "After modifying its length and width, r1 is now ";
    r1.print(); cout << endl;

    system("Pause");
    return 0;
}
```