COMPUTER SCIENCE
CITY COLLEGE OF NEW YORK

CSC212

Data Structure

- Section FG

Lecture 2

ADT and C++ Classes (I)

Instructor: Feng HU

Department of Computer Science

City College of New York

Outline

A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

Object Oriented Programming

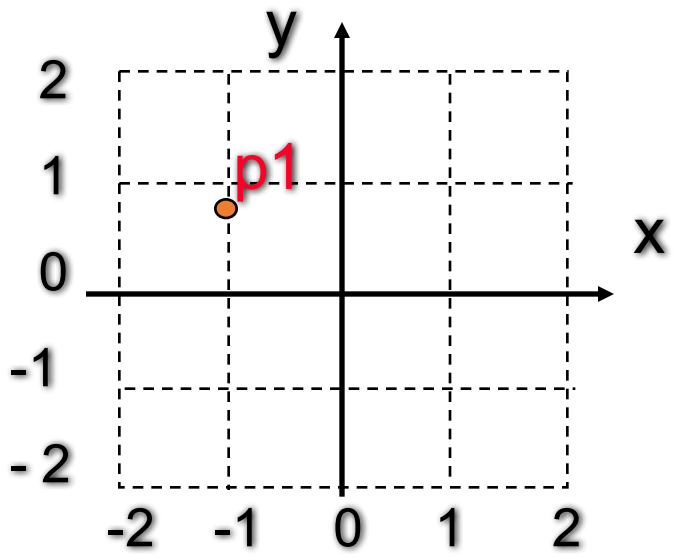
- Chapter 2 introduces Object Oriented Programming.
- OOP is the typical approach to programming which supports the creation of new data types and operations to manipulate those types.
- This lecture gives a review of C++ Classes and introduces ADTs.

C++ Classes and ADTs

- Class
  - Mechanism to create objects and member functions
  - Support information hiding
- Abstract Date Types (ADTs)
  - mathematical data type
  - Class as an ADT that programmers can use without knowing how the member functions are implemented - i.e. with information hiding

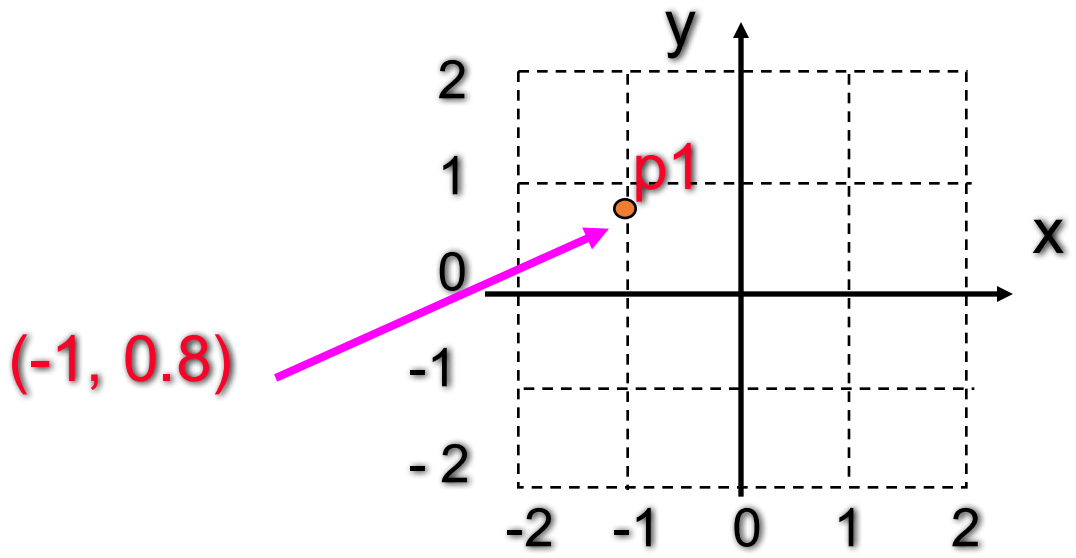
A point ADT

- A data type to store and manipulate a single point on a plane
- Manipulations
  - Initialize
  - Retrieval
  - Shift



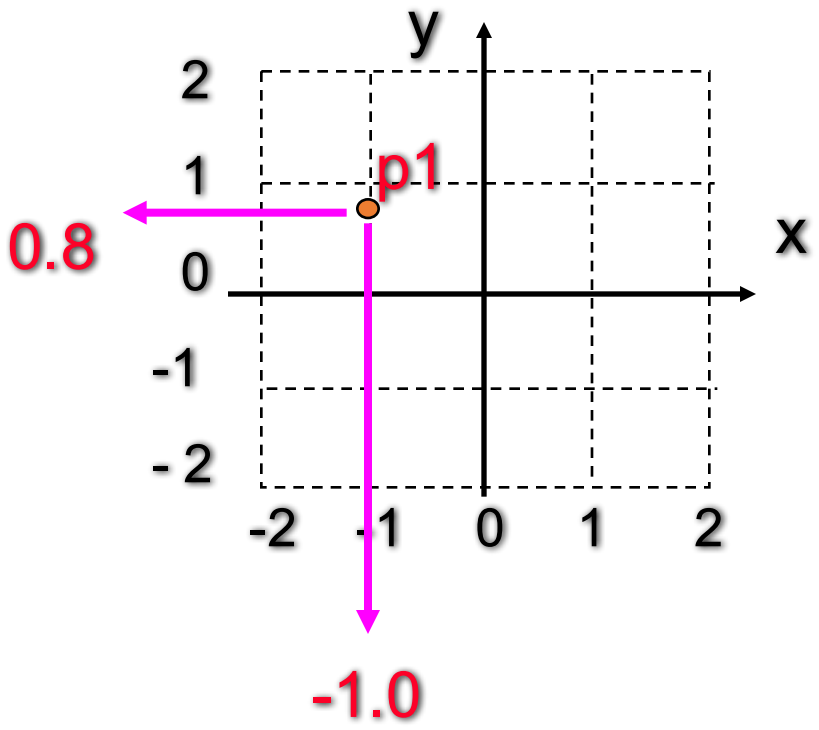
A point ADT

- A data type to store and manipulate a single point on a plane
- Manipulations
  - Initialize
  - Retrieval coordinates
  - Shift



A point ADT

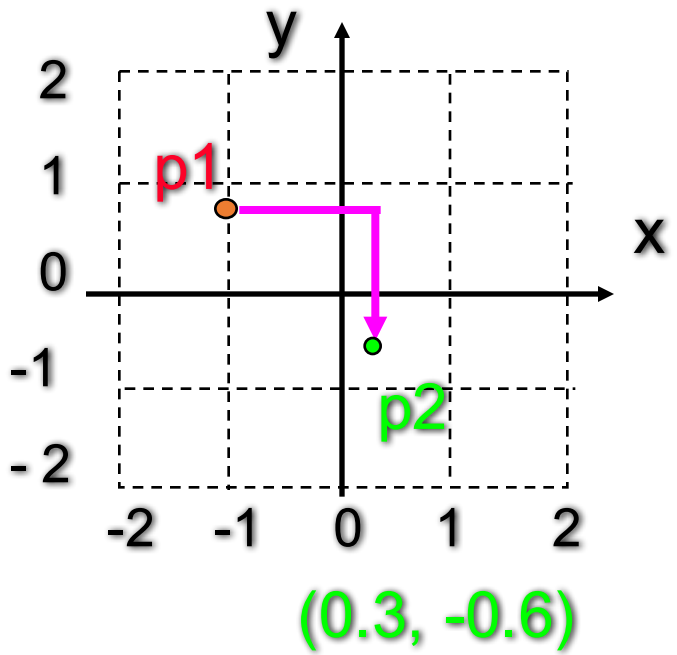
- A data type to store and manipulate a single point on a plane
- Manipulations
  - Initialize
  - Retrieval coordinates
  - Shift



A point ADT

- A data type to store and manipulate a single point on a plane
- Manipulations
  - Initialize
  - Retrieval coordinates
  - Shift by

(1.3, -1.4)



Outline

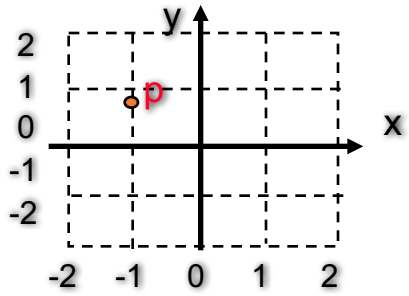
A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

point Definition



- We can implement the point object using a data type called a class.

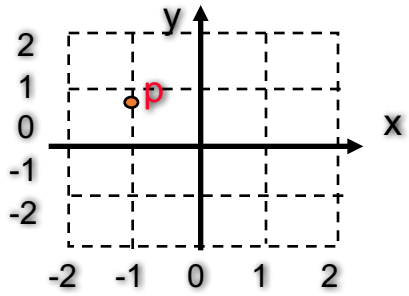
```
class point
{

        . . .


};
```

Don't forget the semicolon at the end

point Definition

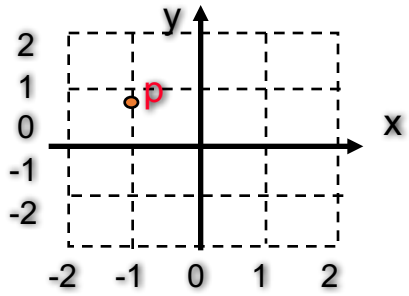


- The class will have two components called x and y. These components are the x and y coordinates of this point.
- Using a class permits two new features . . .

```
class point
{
      . . .
      double x;
      double y;

};
```

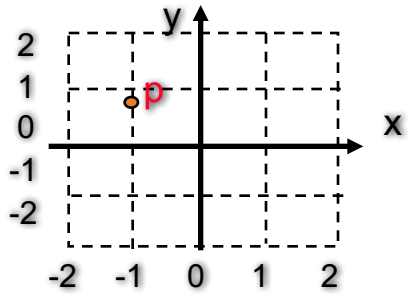
point Definition



❶ The two components will be private member variables. This ensures that nobody can directly access this information. The only access is through functions that we provide for the class.

```
class point
{
      . . .
private:
    double x;
    double y;
};
```

point Definition

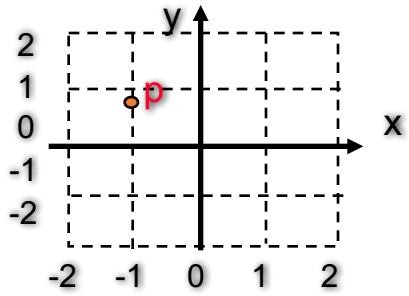


❷ In a class, the functions which manipulate the class are also listed.

```
class point
{
public:
    . . .
private:
    double x;
    double y;
};
```

Prototypes for the point functions go here, after the word public:

point Definition

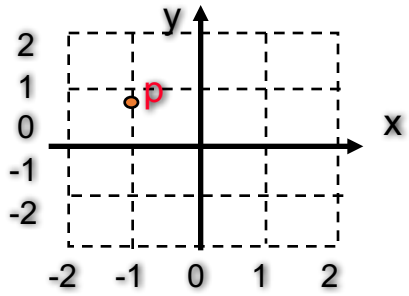


❷ In a class, the functions which manipulate the class are also listed.

```
class point
{
public:
    . . .
private:
    double x;
    double y;
};
```

Prototypes for the point member functions go here

point Definition

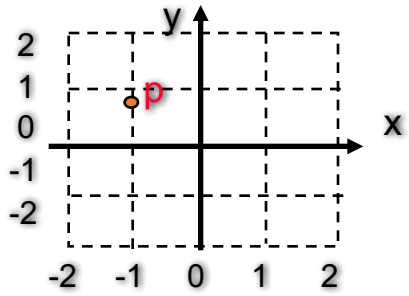


Our point has at least four member functions:

```
class point
{
public:
    void initialize(double init_x, double init_y);
    void shift(double dx, double dy);
    double  get_x() const;
    double  get_y( ) const;
private:
    double x;
    double y;
};
```

Function bodies will be elsewhere.

point Definition

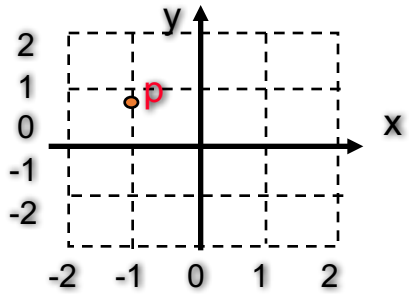


The keyword **const** appears after two prototypes:

```
class point
{
public:
    void initialize(double init_x, double init_y);
    void shift(double dx, double dy);
    double get_x( ) const;
    double get_y( ) const;
private:
    double x;
    double y;
};
```

This means that these functions will not change the data stored in a point ADT.

Files for the **point** ADT



- The point class definition, which we have just seen, is placed with documentation in a file called point.h, outlined here.
- The implementations of the four member functions will be placed in a separate file called point.cxx, which we will examine in a few minutes.

Documentation: (Preconditions and Postconditions)

Class definition:
- point class definition which we have already seen

Outline

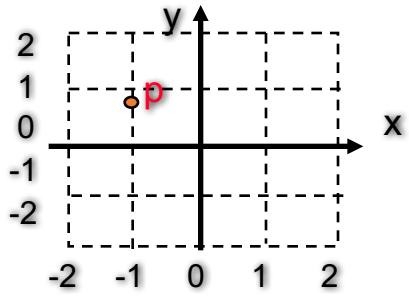
A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

Using the **point** ADT

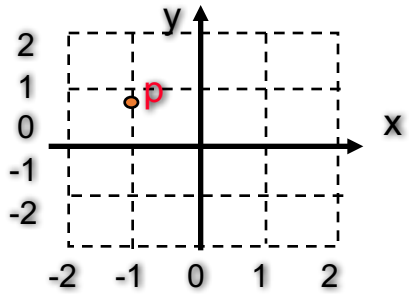


- A program that wants to use the point ADT must **include** the point.h header file (along with its other header inclusions).
- File pointmain1.cxx

```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

...
```

Using the **point** ADT

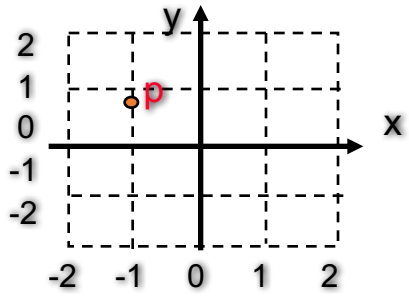


- Just for illustration, the example program will declare two point variables named p1 and p2.

```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

int main( )
{
    point p1;
    point p2;
```

Using the **point** ADT



- Just for illustration, the example program will declare two point objects named p1 and p2.

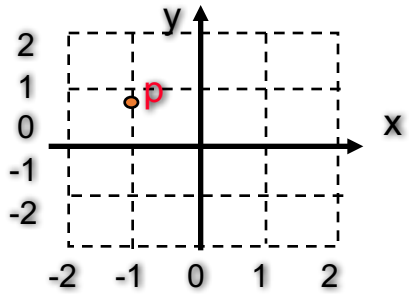
```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

int main( )
{
    point p1;
    point p2;



 }
```

Using the **point** ADT

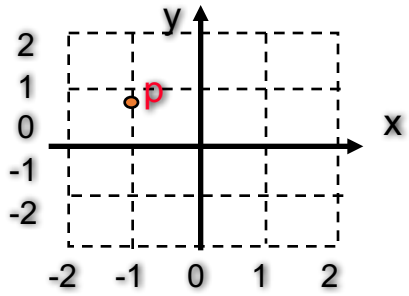


- The program starts by calling the initialize member function for p1.

```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

int main( )
{
    point p1;
    point p2;
    p1.initialize(-1.0, 0.8);
```

Using the **point** ADT



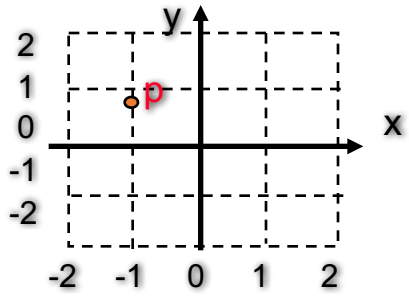
- The program starts by activating the initialize member function for p1.

```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

int main( )
{
    point p1:
    point p2;

    p1.initialize(-1.0,  0.8);
```

Using the **point** ADT



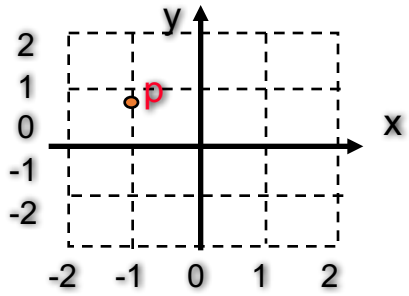
❶The member function activation consists of four parts, starting with the object name.

```
int main( )
 {
    point p1;
    point p2;

    p1.initialize(-1.0, 0.8);
```

Name of the object

Using the **point** ADT



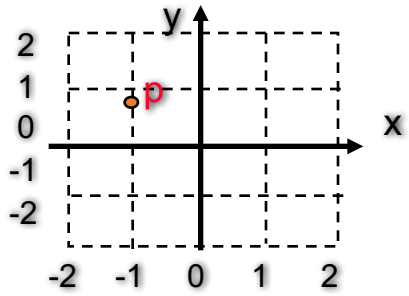
❷The instance (object) name is followed by a period.

```
int main( )
{
    point p1;
    point p2;

    p1.initialize(-1.0, 0.8);
```

A Period

Using the **point** ADT



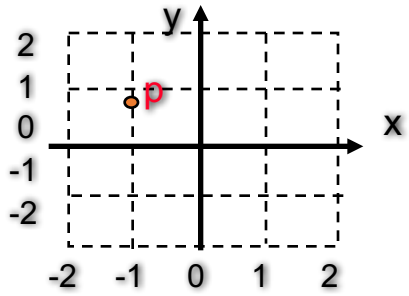
❸After the period is the name of the member function that you are activating.

```
int main( ) {
    point p1;
    point p2;

    p1.initialize(-1.0, 0.8);
```

Name of the Function

Using the **point** ADT



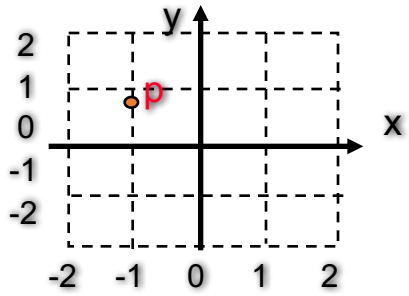
❹Finally, the arguments for the member function. In this example the first argument (x coordinate) and the second argument (y coordinate)

```
int main( ) {
    point p1;
    point p2;

    p1.initialize(-1.0, 0.8);
```

Arguments

A Quiz



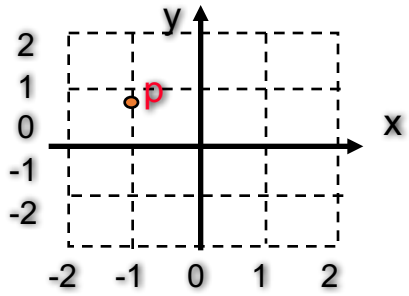
How would you activate p1's get_x member function ?

What would be the output of p1's get_x member function at this point in the program ?

```
int main( )
{
    point p1;
    point p2;

    p1.initialize(-1.0,  0.8);
```

A Quiz



Notice that the **get_x** member function has no arguments.

At this point, activating **p1.get_x** will return a double value

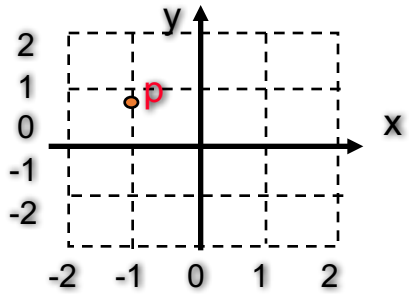
-1.0.

```
int main( ) {
    point p1;
    point p2;

    p1.initialize(-1.0,  0.8);

    cout << p1.get_x( ) <<endl;
```

A Quiz



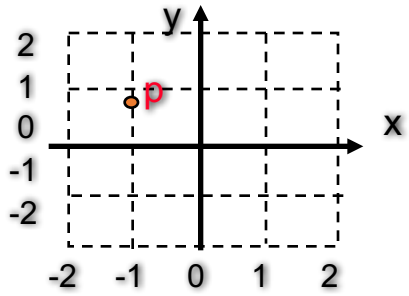

```
int main( )
{
    point p1;
    point p2;

    p1.initialize(-1.0,  0.8);
    cout << p1.get_x( ) << p1.get_y() << endl;
    p2.initialize(p1.get_x(),  p1.get_y());
    cout << p2.get_x( ) << p2.get_y() << endl;
    p2.shift(1.3, -1.4);
    cout << p2.get_x( ) << p2.get_y() << endl;

    . . .
```

Trace through this program, and tell me the complete output.

A Quiz




```
int main( )
{
    point p1;
    point p2;

    p1.initialize(-1.0,  0.8);
    cout << p1.get_x( ) << p1.get_y() << endl;
    p2.initialize(p1.get_x(),  p1.get_y());
    cout << p2.get_x( ) << p2.get_y() << endl;
    p2.shift(1.3, -1.4);
    cout << p2.get_x( ) << p2.get_y() << endl;

    . . .
```

-1.0 0.8

-1.0 0.8

 0.3 -0.6

What you know about Objects

- ✓ Class = Data + Member Functions.
- ✓ You know how to define a new class type, and place the definition in a header file.
- ✓ You know how to use the header file in a program which declares instances of the class type.
- ✓ You know how to activate member functions.
- ✖ But you still need to learn how to write the bodies of a class's member functions.

Outline

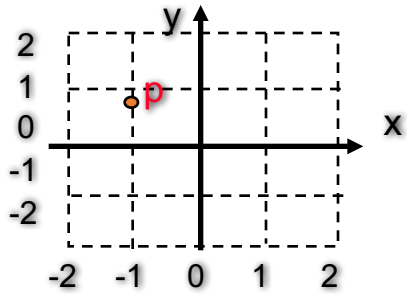
A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

point Implementation

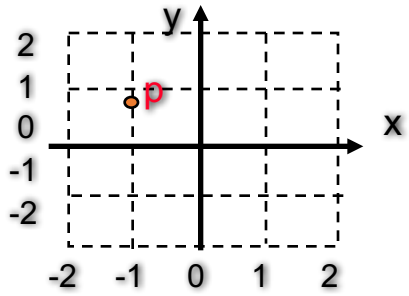


Remember that the member function's bodies generally appear in a separate point.cxx file.

```
class point
{
public:
    void initialize(double init_x, double init_y);
    void shift(double dx, double dy);
    double get_x( ) const;
    double get_y( ) const;
private:
    double x;
    double y;
};
```

Function bodies will be in .cxx file.

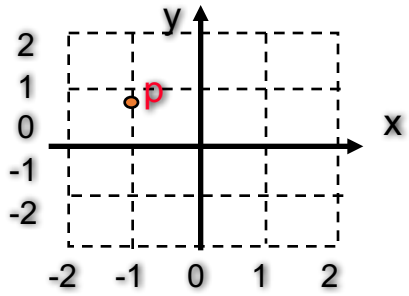
point Implementation



We will look at the body of intialize, which must assign its two arguments to the two private member variables.

```
class point
{
public:
    void initialize(double init_x, double init_y);
    void shift(double dx, double dy);
    double get_x( ) const;
    double get_y( ) const;
private:
    double x;
    double y;
};
```

point Implementation

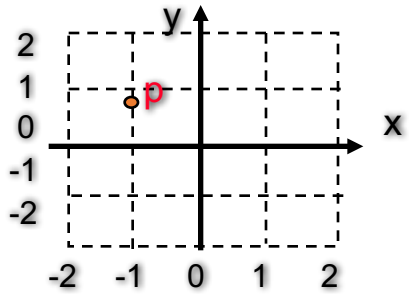


For the most part, the function's body is no different than any other function body.

```
void point::initialize(double init_x, double init_y)
{
    x = init_x;
    y = init_y;
}
```

But there are two special features about a member function's body . . .

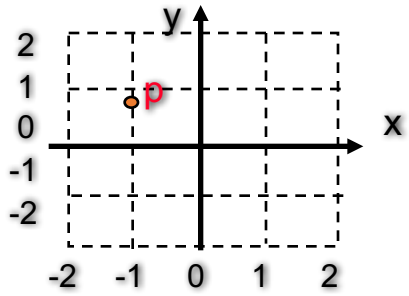
point Implementation



❶In the heading, the function's name is preceded by the class name and :: - otherwise C++ won't realize this is a class's member function.

```
void point::initialize(double init_x, double init_y)
{
    x = init_x;
    y = init_y;
}
```

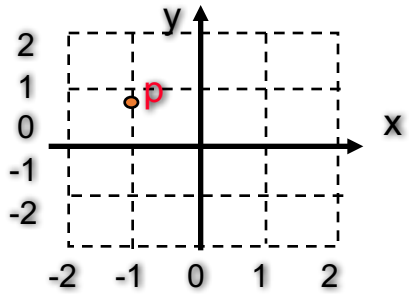
point Implementation



❷Within the body of the function, the class's member variables and other member functions may all be accessed.

```
void point::initialize(double init_x, double init_y)
{
    x = init_x;
    y = init_y;
}
```

point Implementation



❷Within the body of the function, the class's member variables and other member functions may all be accessed.

```
void point::initialize(double init_
{
    x = init_x;
    y = init_y;
}
```

But, whose member variables are these? Are they

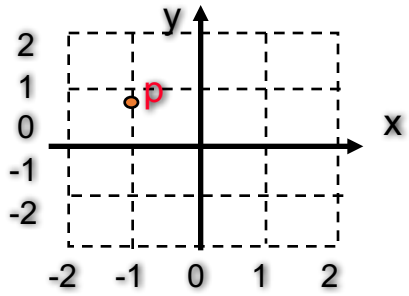
p1.x

p1.y

p2.x

p2.y

point Implementation



❷Within the body of the function, the class's member variables and other member functions may all be accessed.

```
void point::initialize(double init_
{
    x = init_x;
    y = init_y;
}
```

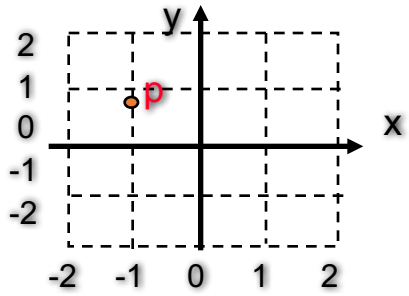
If we activate

p1.initialize:

p1.x

p1.y

point Implementation



❷Within the body of the function, the class's member variables and other member functions may all be accessed.

```
void point::initialize(double init_
{
    x = init_x;
    y = init_y;
}
```

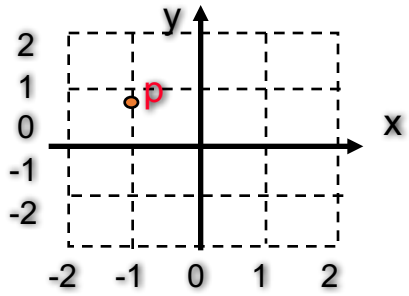
If we activate

p2.initialize:

p2.x

p2.y

point Implementation



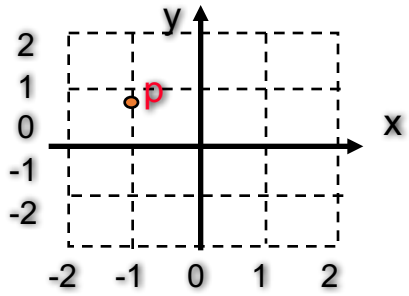
Here is the implementation of the get_x member function, which return the x coordinate:

```
double point::get_x() const
{

    return x;

}
```

point Implementation



Here is the implementation of the get_x member function, which return the x coordinate:

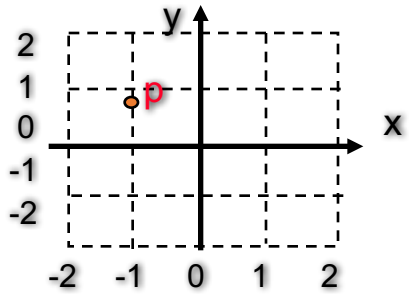
```
double point::get_x() const
{

    return x;

}
```

Notice how this member function implementation uses the member variable x of the point object.

point Implementation

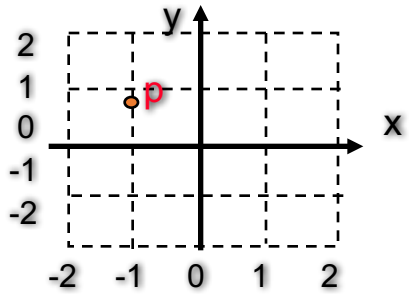


Member functions may activate other member functions

```
void point::origin()
{
    x = 0.0;
    y = 0.0;
}
```

Notice this member function implementation still directly assign the member variables x and y.

point Implementation



Member functions may activate other member functions

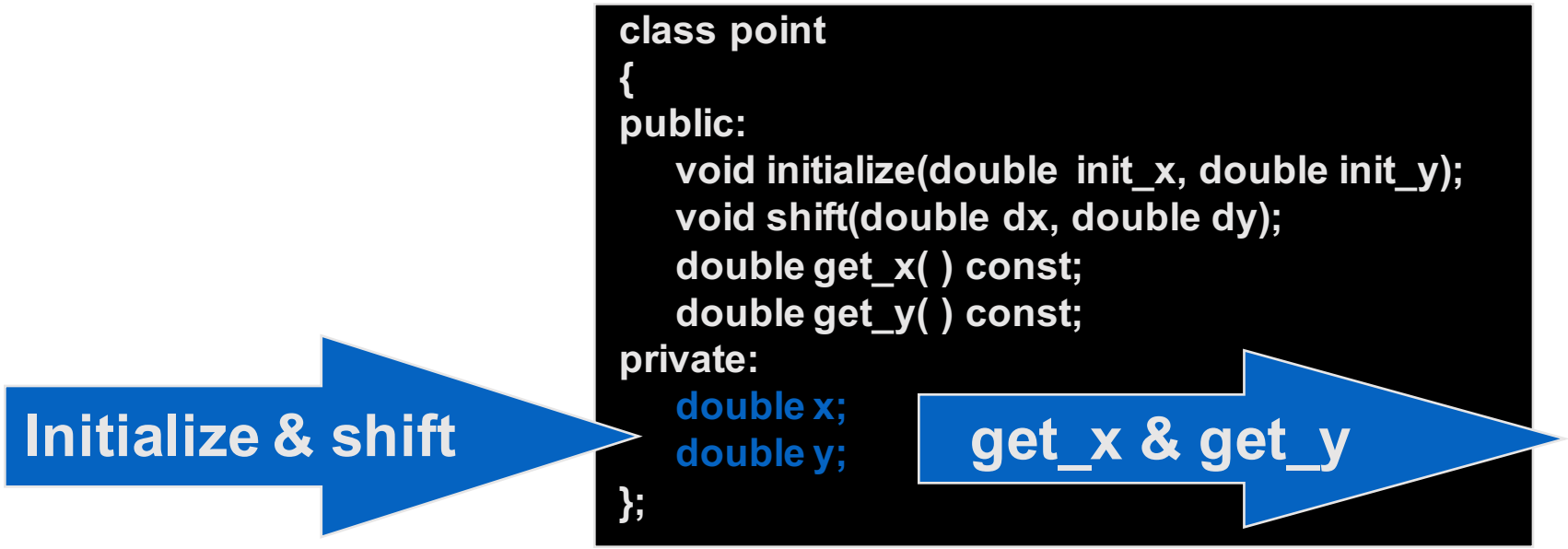
```
void point::origin()
{
    initialize(0.0, 0.0);
}
```

Without object name

Notice how this member function implementation uses the member function initialize.

A Common Pattern

- Often, one or more member functions will place data in the member variables...



- ...so that other member functions may use that data.

Summary of classes

- Classes have member variables and member functions. An object is a variable where the data type is a class.
- You should know how to declare a new class type, how to implement its member functions, how to use the class type.
- Frequently, the member functions of an class type place information in the member variables, or use information that's already in the member variables.
- Next we will see more features of OOP and classes.

Assignments

- Reading:
  - Chapter 2.3-2.5
- Programming assignment 1 - Due Wed, Sep. 16
  - Need all of chapter 2 to finish, but you can start doing it now
  - Requirements and guidelines have been posted on the course web site
- C++ Installation Guide online
  - Linux Users: See the assignment #1 guidelines
  - Mac/Win Users: Check the course bulletin (by Wai Khoo)

Break

Outline

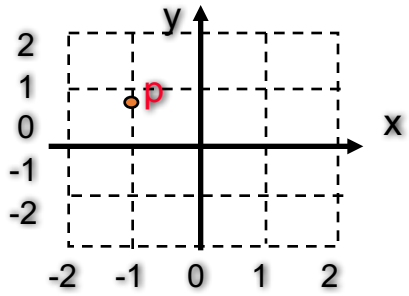
A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

Constructors: point Initialization



- The program starts by activating the initialize member function for p1.

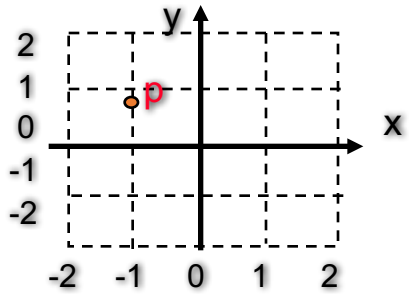
```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

int main( )
{
    point p1:
    point p2;

    p1.initialize(-1.0,  0.8);
```

First improvement: automatic initialization without activating the initialize function

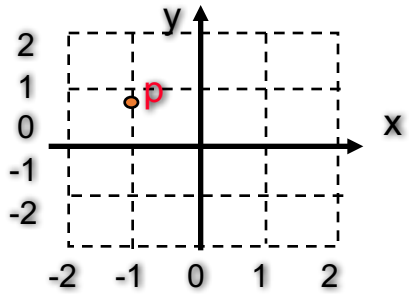
Constructors: point Initialization



We can provide a normal member function initialize

```
class point
{
public:
    void initialize(double init_x, double init_y);
    void shift(double dx, double dy);
    double  get_x() const;
    double  get_y( ) const;
private:
    double x;
    double y;
};
```

Constructors: point Initialization



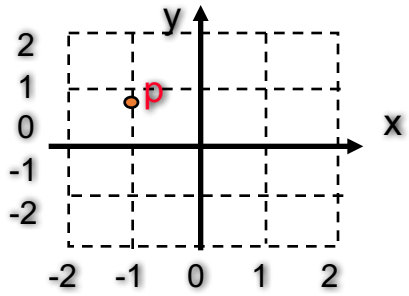
Or use a constructor that is automatically called

```
class point
{
public:
    point(double init_x, double init_y);
    void shift(double dx, double dy);
    double get_x() const;
    double get_y( ) const;
private:
    double x;
    double y;
};
```

-function name same as class name

- no return type, even no "void" !

Constructors: Implementation

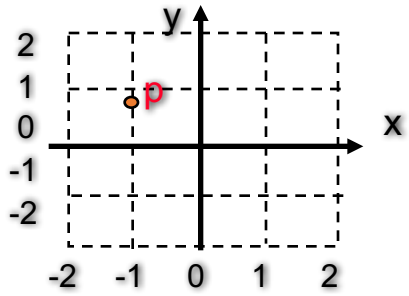


For the most part, the constructor is no different than any other member functions.

```
void point::initialize(double init_x, double init_y)
{
    x = init_x;
    y = init_y;
}
```

We only need to replace initialize with point

Constructors: Implementation



For the most part, the constructor is no different than any other member functions.

```
point::point(double init_x, double init_y)
{
    x = init_x;
    y = init_y;
}
```

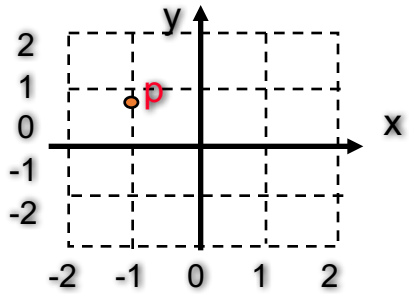
But there are three special features about constructor .

. .

Constructors

- Constructor is a member function which
  - the name must be the same as the class name
  - automatically called whenever a variable of the class is declared
  - arguments must be given after the variable name (when declared in user file)
- A way to improve the initialize function
  - by providing an initialization function that is guaranteed to be called

Constructors: point Initialization



- Automatically called when declared.
- Parameters after the object names

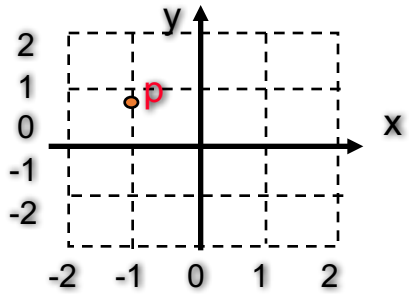
```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

int main( )
{
    point p1:
    point p2;

    p1.initialize(-1.0,  0.8);
```

First improvement: automatic initialization without explicitly activating an initialize function

Constructors: point Initialization



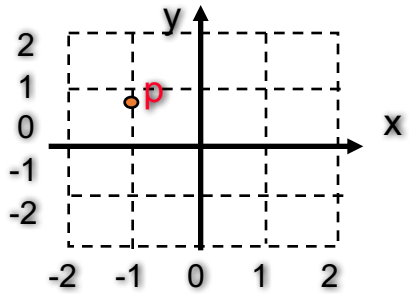
- Automatically called when declared.
- Parameters after the object names

```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

int main( )
{
    point p1(-1.0,  0.8):
    point p2(0.3, 0.6);
```

First improvement: automatic initialization without explicitly activating an initialize function

Default Constructors



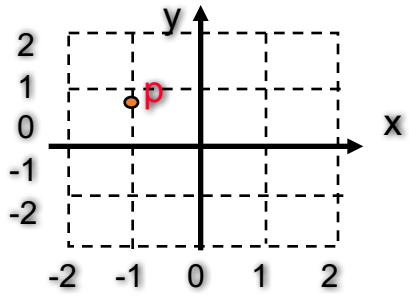
- Automatically called when declared.
- Parameters after the object names

```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

int main( )
{
    point p1(-1.0,  0.8):
    point p2(0.3, 0.6);
```

Sometime we want to define an object with no parameters…

Default Constructors



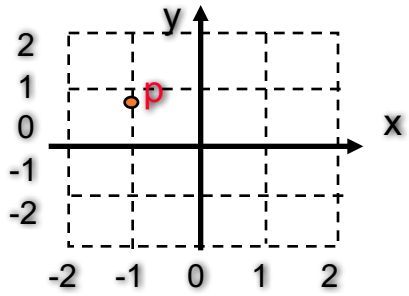
- Automatically called when declared.
- NO parameters after the object name p2

```
#include <iostream.h>
#include <stdlib.h>
#include “point.h"

int main( )
{
    point p1(-1.0,  0.8);
    point p2;
```

…not even a pair of parentheses

Default Constructors



We could provide a second constructor with no parameters

```
class point
{
public:
    point();
    point(double init_x, double init_y)
    …
private:
    double x;
    double y;
};
```

Implementation

```
point::point()
{
    x = 0.0;
    y = 0.0;
}
```

Constructors: Function Overloading

- You may declare as many constructors as you like – one for each different way of initializing an object
- Each constructor must have a distinct parameter list so that the compiler can tell them part
- Question: How many default constructor is allowed?

Constructors: automatic default constructor

- What happens if you write a class without any constructors?
- The compiler automatically creates a simple default constructor
  - which only calls the default constructors for the member variables that are objects of some other classes
- Programming Tip :Always provide your own constructors, and better with a default constructor

Value Semantics of a Class

- Value semantics determines how values are copied from one object to another
- Consists of two operations in C++
  - The assignment operator
  - The copy constructor
- Document the value semantics
  - When you implement an ADT, the document should include a comment indicating that the value semantics is safe to use.

Value Semantics: assignment operator

- Automatic assignment operator
  - For a new class, C++ normally carries out assignment by simply copying each variable from the object on the right to that on the left
  - our new class point can use automatic assignment operator

```
point p1(-1.0,  0.8), p2;

p2 =  p1;

cout << p2.get_x() <<“ “ << p2.get_y();
```

- When automatic assignment fails
  - we will see examples in Lecture 4 (pointers and dynamic arrays)

Value Semantics: copy constructor

- A copy constructor
  - is a constructor with exactly one parameter whose data type is the same as the constructor's class
  - is to initialize a new object as an exact copy of an existing object
- An example

```
point p1(-1.0,  0.8);
point p2 (p1);
cout << p2.get_x() <<" " << p2.get_y();
```

Value Semantics: copy constructor

- A copy constructor
  - is a constructor with exactly one parameter whose data type is the same as the constructor's class
  - is to initialize a new object as an exact copy of an existing object
- An alternative syntax

```
point p1(-1.0,  0.8);
point p2 = p1;
cout << p2.get_x() <<" "<< p2.get_y();
```

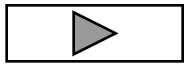
Value Semantics: discussion

- point p2 = p1; versus p2 = p1;
  - The assignment p2 = p1; merely copies p1 to the already existing object p2 using the assignment operator.
  - The syntax point p2 = p1; looks like an assignment statement, but actually a declaration that both declare a new object, and calls the copy constructor to initialize p2 as a copy of p1.

- p2 will be the same iff the assignment operator and the copy constructor do the same things

Copy Constructor: Implementation

- You may write a copy constructor much like any other constructor
  - Lecture 4 and later



- Take advantage of a C++ feature
  - automatic copy constructor
  - similar to assignment, the automatic copy constructor initializes a new object by merely copy all the member variables from the existing object.
  - Automatic versions may fail!

Point Demo

Constructors, etc.– a summary

- Constructor is a member function
  - define your own constructors (including a default)
  - automatic default constructor

  inline member functions (Ch 2.2)
  - Place a function definition inside the class definition
  - for time efficiency
- value semantics of a class
  - assignment operators and copy constructor
  - automatic assignment op and copy constructor

Outline

A Review of C++ Classes (Lecture 2)

- OOP, ADTs and Classes
- Class Definition, Implementation and Use
- Constructors and Value Semantics

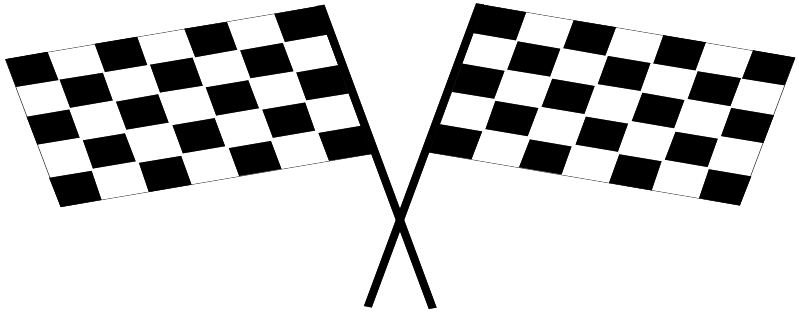
More on Classes (Lecture 3)

- Namespace and Documentation
- Classes and Parameters
- Operator Overloading

Assignments

- Reading:
  - Chapter 2.3-2.5
- Programming assignment 1 - Due Sep. 14
  - Need all of chapter 2 to finish, but you can start doing it now
  - Requirements and guidelines will be posted on the course web site

The first part (p.3-47) of this lecture was adapted from:



THE END