Introduction to Programming Using

C/C++

Presented By
Dr. Eman Hesham, DBA

Java™ Education and Technology Services

Invest In Yourself , Develop Your Career

Java™ Education and Technology Services

Chapter 5

Object-Oriented programming Relations

Chapter 5 Outline

1. Collections of objects
   1. Static Allocations : array of objects
   2. Dynamic Allocation: pointer to object
2. Class Relations
   1. Association
   2. Aggregation
   3. Composition
3. Example

Collections of objects

- We can deal with objects :
  - we may create an array of objects,
  - we may make dynamic memory allocation with objects,
  - we may initialize the array of objects.

Collections of objects

- Static Allocations : array of objects
  - an array of objects exactly

    Complex carr[10];

  - The above line is declare an array of 10 Complex objects and each object call the default constructor
  - may call a different constructor for each object as we need:

    Complex carr[3] = {Complex(2, 4), Complex(),Complex(8)};

  - If we write constructors to some objects and not to all, the remaining will call the default constructor
  - Like any array in C, we may not write the size in array declaration if we make initialization with constructors.

Collections of objects

- Static Allocations : array of objects

```
int main()
{
    Complex arr[3] = {Complex(2), Complex(), Complex(5,7)};
    for(int i = 0 , i<3 ; i++)
        arr[i].printComplex();
    getch();
    return 0;
}
```

Collections of objects

- Dynamic Allocation: pointer to object
  - use a pointer to object to make dynamic memory allocation with number of objects allocated in the heap memory and deal with them like array.

Complex * cptr;

- cptr = new Complex(2.1, 7.3); // just allocate one Complex
- Cptr = new Complex[12]; // allocate 12 complex contiguous

  // but with default constructor only

Class Relations

Class Relations

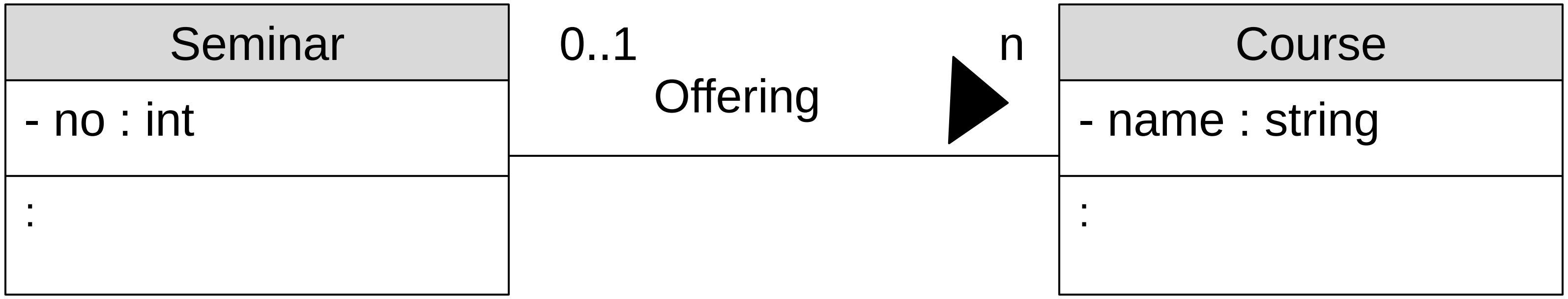
I. Association

- Is a relation between two classes.
- It allows one object instance to “use” another to perform an action on its behalf.
- Two objects have their own lifecycle and there is no owner .
- Both can created and deleted independently.
- **Examples:**
  - Students are “on waiting list “ for a seminar.
  - Professors “instruct” a seminar.
  - Seminar is an “offering” of courses

Class Relations

I. Association

- **Examples:**
  - Seminar is an “offering” of courses



  - How many courses does seminar offer? None, one , many?

0..1	**1**	**0..***	**1..***	**n**	**0..n**	**1..n**

Class Relations

I. Association

- **Examples:**
  - class **Course**{

```
:
Course();
}
```

  - class **Seminar**{

```
        Course * x;
public:
        Seminar();
        void offer ( Course *c);
}
```

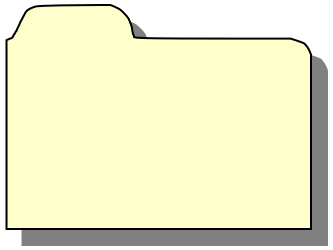
```
int main() {
        Course c1;
        Seminar s1;
        s1.offer(&c1);
```

ClassA may be linked to ClassB in order to show that one of its methods include a reference to the another one as a parameter

Class Relations

I. Association

- Create a folder called “ links” .
- Create shortcuts inside this folder to Google , YouTube and Facebook
- Delete this folder.
- Open your browser to check Google , YouTube and Facebook are still exit or not



- **No owner :** Association is a relation where all the object have different lifecycle.

Class Relations

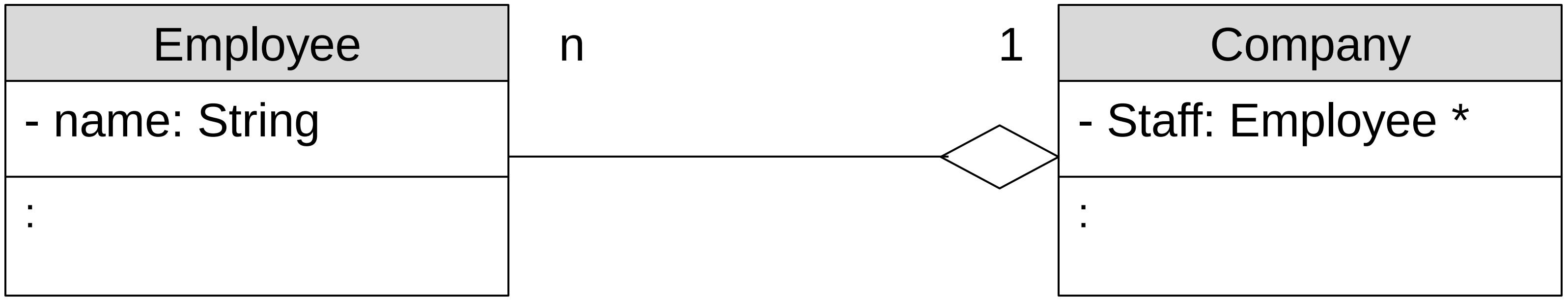
II. Aggregation

- Is a special type of association. [strong Association]
- Object of one class " has" an object of the another one
- Two objects have their own lifecycle and there is one owner at atime .
- Both can created and deleted independently.
- **Examples:**
  - Room contains many Tables.

Class Relations

II. Aggregation

- **Examples:**
  - A single Employee can not belong to multiple companies .



Class Relations

II. Aggregation

- **Examples:**

```
– class Employee{
:
Employee();
}
– class Company{
      Employee* staff;
public:
      Company (Employee *x){
            staff= x;
      }
}
```

```
int main() {
      Employee emp;
      Company c1(&emp);
```

ClassA may be linked to **ClassB** in order to show that one of its constructors include a reference to the another one as a parameter

Class Relations

II. Aggregation

- Create a file called “ file.txt” .
- Make a simple Application to open the file.txt in rw mode.
- Run an instance of the application.
- Try to run another instance of this application.
- Sure the application and the file has a separated lifecycles .
- However this file can be opened only by one application. One parent at a time.

- **All Aggregations are associations but not all associations are aggregations**

Class Relations

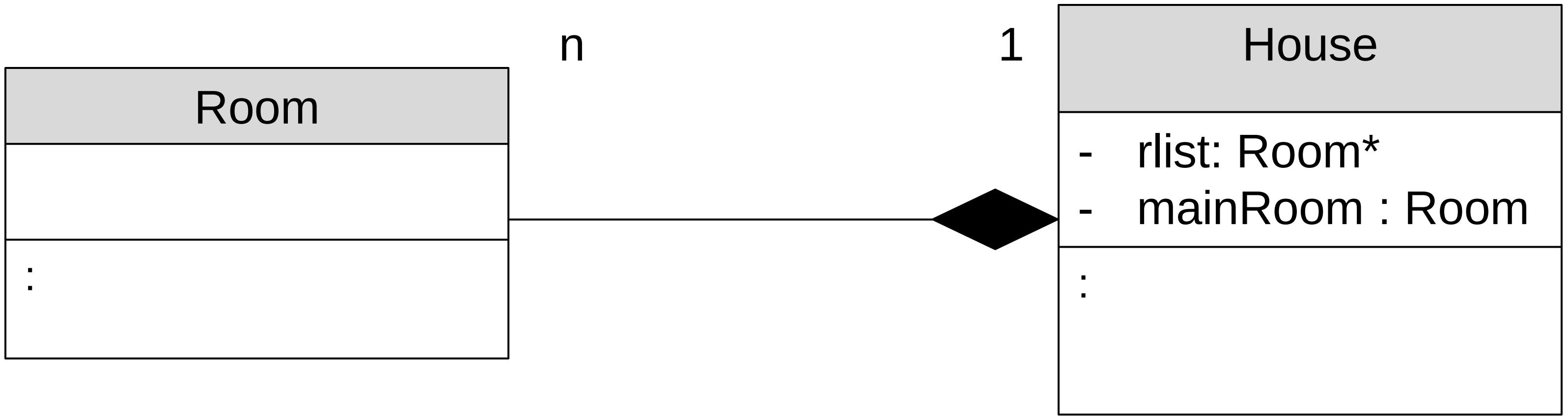
III. Composition

- Is a Strong type of Aggregation.
- Part object does not have its own lifecycle.
- If the whole object gets deleted , all of its parts will also deleted.
- Typically use normal member variables
- Can use pointer values if the composition class automatically handle allocation / de-allocation of these values
- **Examples:**
  - House contains multiple Rooms.
  - Any Room can not be belong to two Houses
  - If we delete the house , all rooms will be deleted

Class Relations

III. Composition

- **Examples:**



Class Relations

III. Composition

- **Examples:**
  - class **Room**{

```
:
Room();
}
```

  - class **House**{

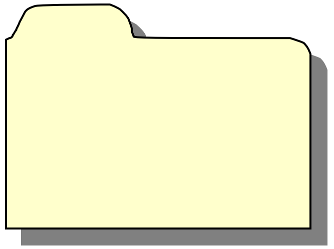
```
        Room* rlist;
        Room mainRoom;
public:
        House():mainRoom() {
                        rlist= new Room[4];
        }
}
```

```
int main() {
            House H;
```

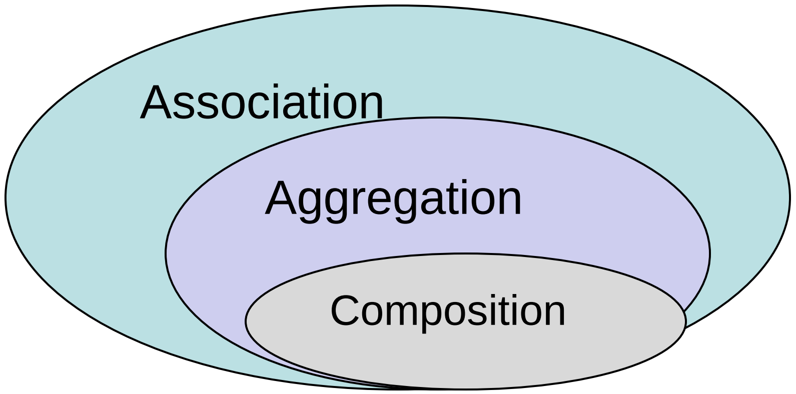
Class Relations

III. Composition

- Create a folder called “ mainFolder” .
- Create two word files inside this folder to todoList and contacts
- Delete this folder.

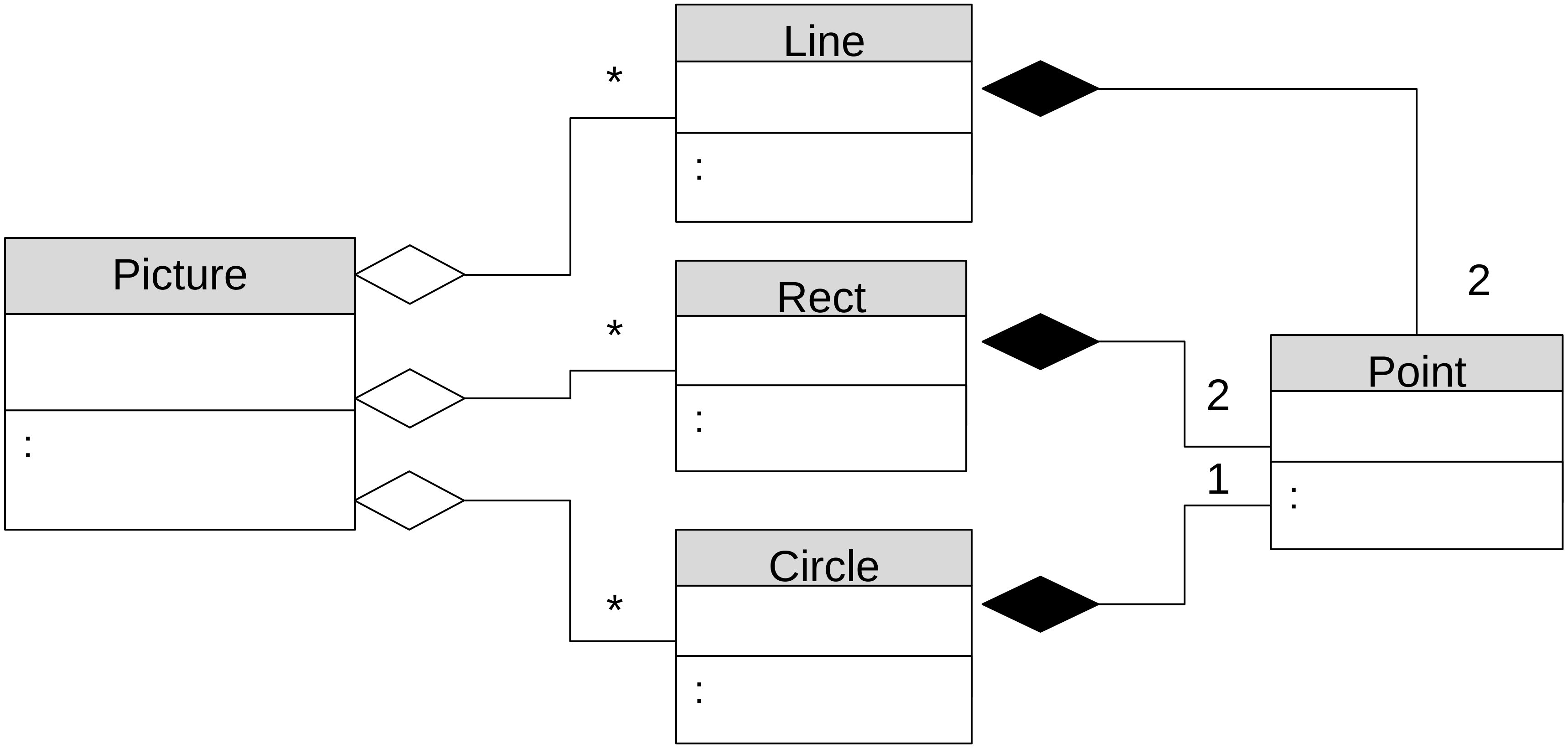


- **All Compositions are Aggregations but not all Aggregations are Compositions**

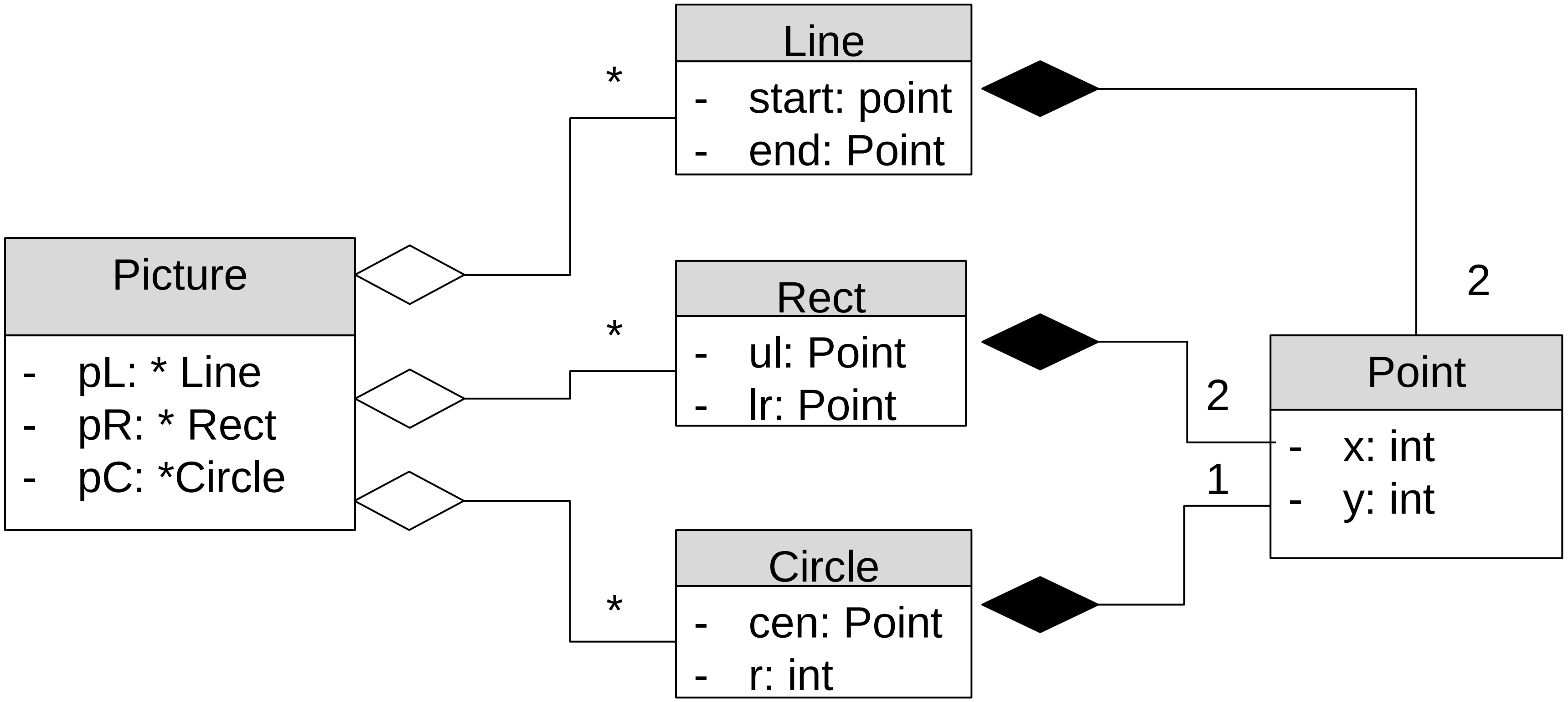


Example

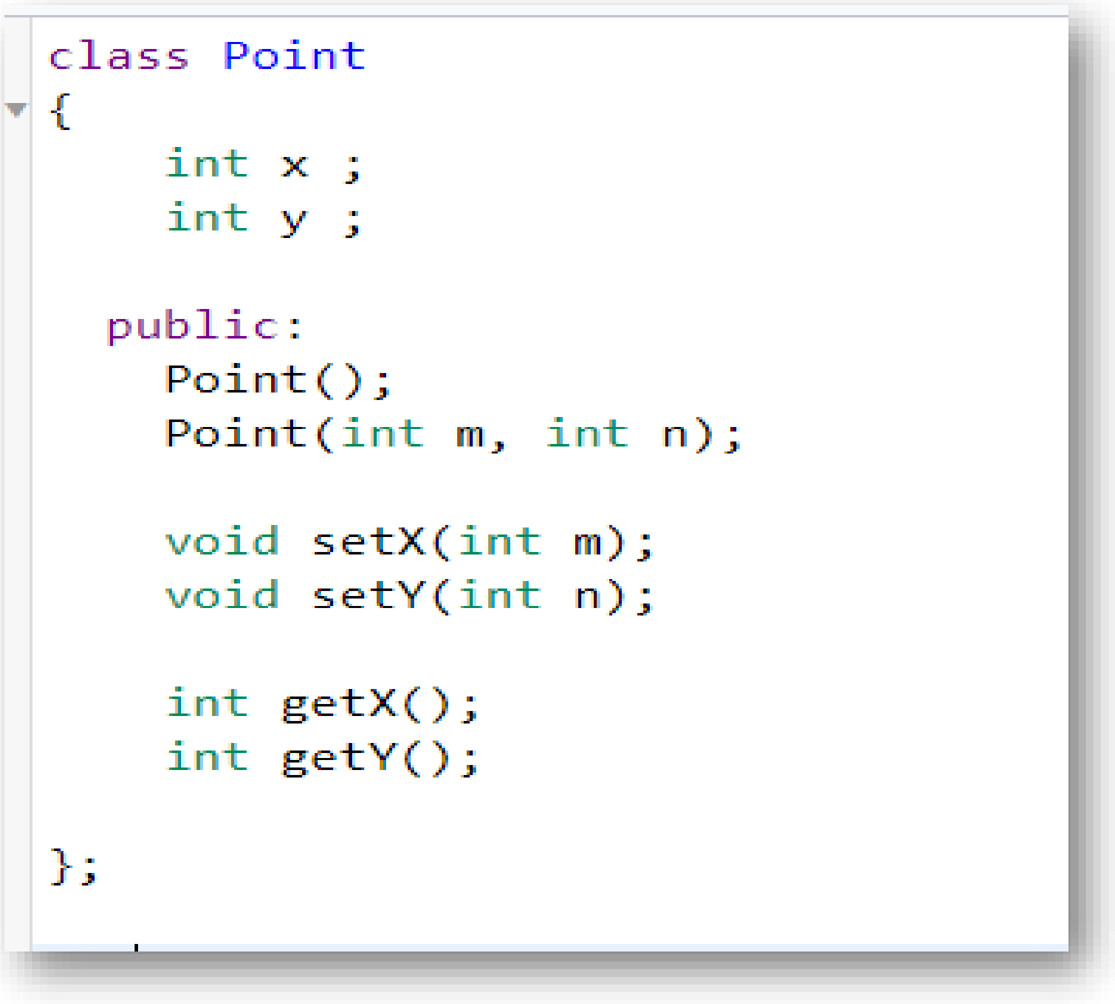
- Create an application to draw a picture of lines , Rectangles and circles.
- Point is the main component of all shapes



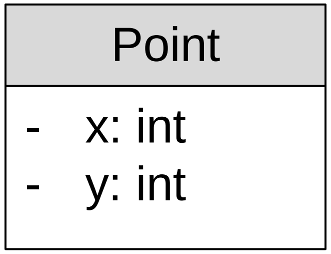
Example


Line
- start: point
- end: Point
Picture
- pL: * Line
- pR: * Rect
- pC: *Circle
Rect
- ul: Point
- lr: Point
Circle
- cen: Point
- r: int
Point
- x: int
- y: int
*
*
*
2
2
1

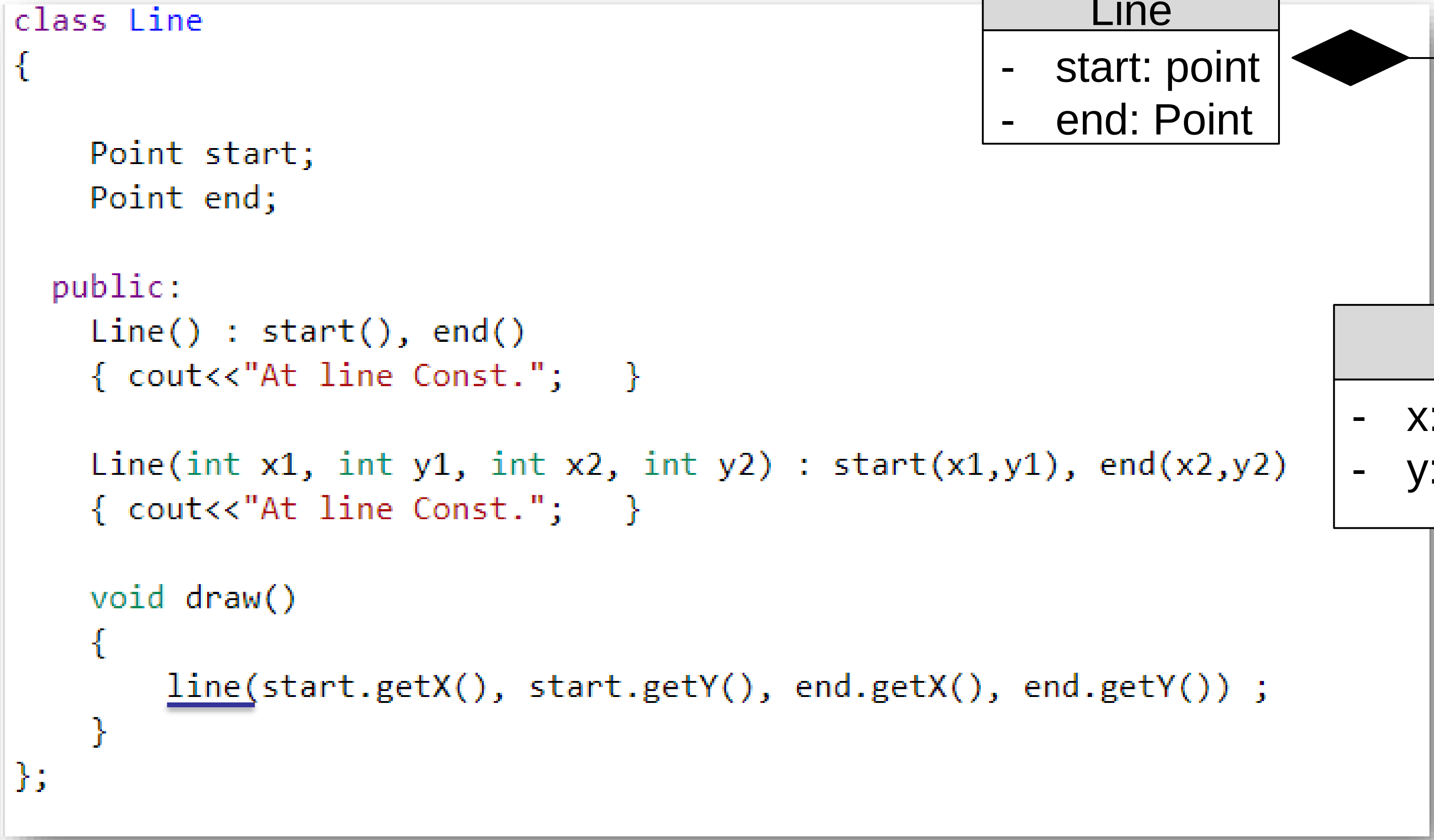
Example



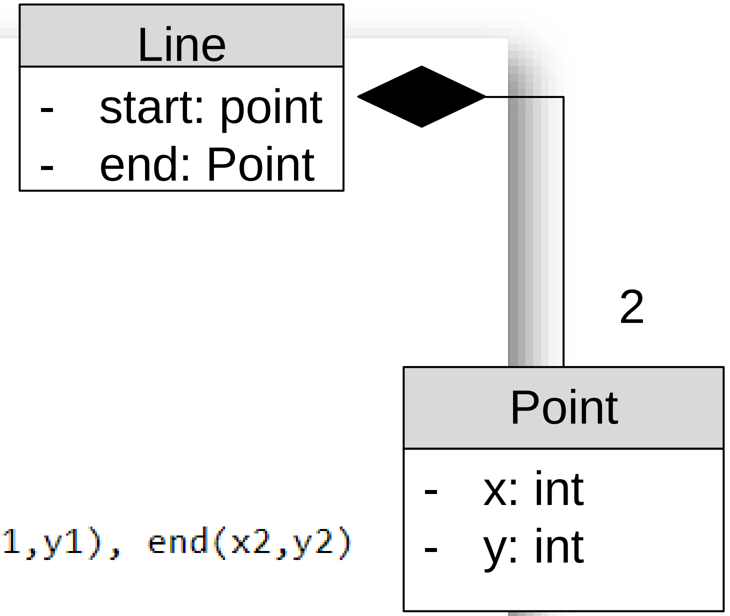
```
class Point
{
    int x ;
    int y ;

  public:
    Point();
    Point(int m, int n);

    void setX(int m);
    void setY(int n);

    int getX();
    int getY();

};
```



Example



```
class Line
{

    Point start;
    Point end;

  public:
    Line() : start(), end()
    { cout<<"At line Const.";   }

    Line(int x1, int y1, int x2, int y2) : start(x1,y1), end(x2,y2)
    { cout<<"At line Const.";   }

    void draw()
    {
        line(start.getX(), start.getY(), end.getX(), end.getY()) ;
    }
};
```



Constructor and destructor Chaining

- In fact, to create an object from class Line, all the instance attributes must be created and allocated in the memory before any constructor of the Line could be called.
- So, the two objects from class Point, start and end, are completely created and allocated as apart of Line object.
- And vise versa, when trying to remove a Line object from the memory, the destructor of Line is the last behavior of the lifetime of the Line object, after it is executed, the Line object is starting to remove from the memory; i.e. its components will be removing, so the two Point objects will be removing, then the destructor for these objects is calling for each one after destructor of Line.

Constructor and destructor Chaining

- The constructor chaining of the an object which has an embedded object, the constructor of the embedded object is execute first and then the constructor of the container object.

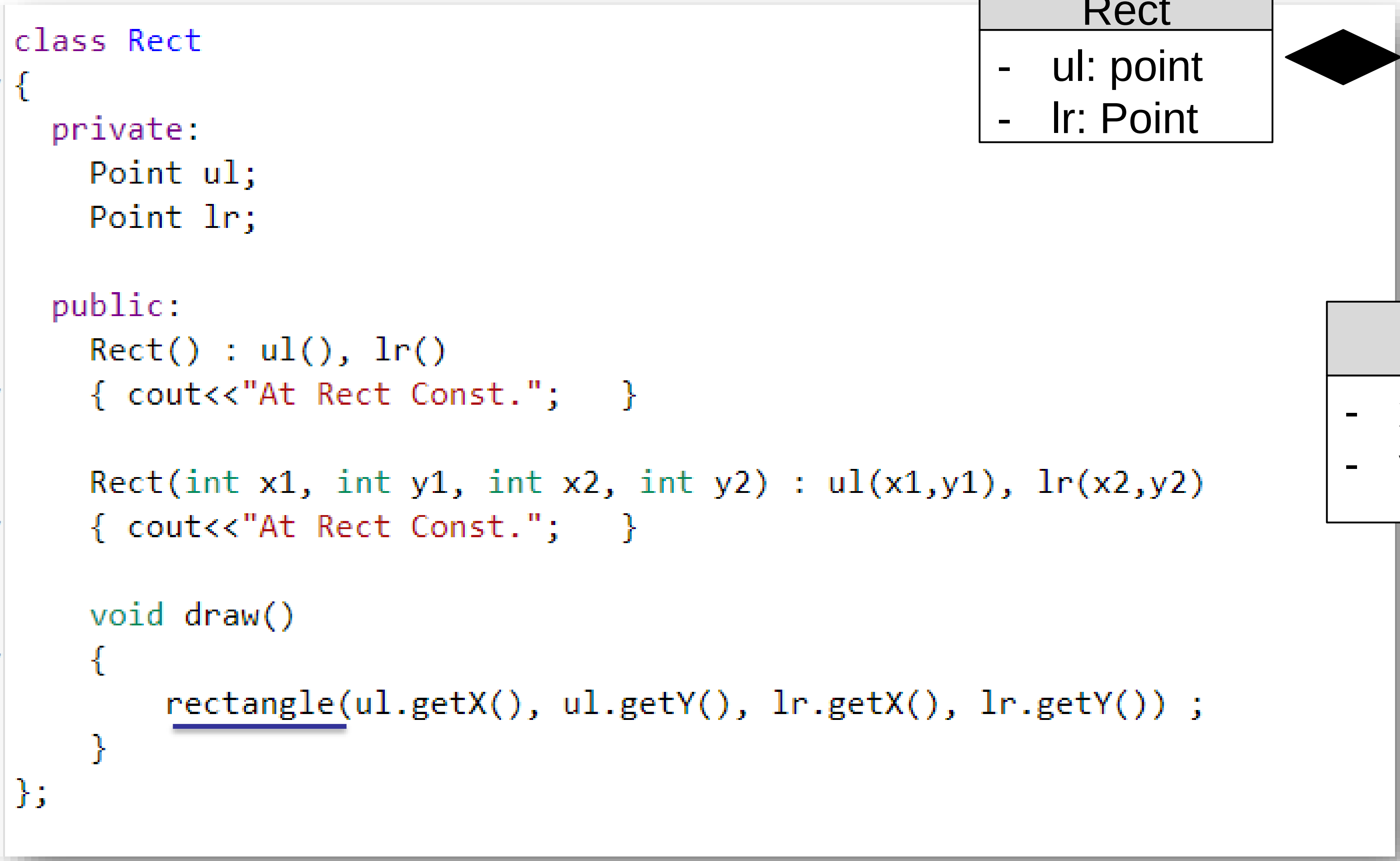
- The destructor chaining of the an object has embedded another object, the destructor of the embedded object is execute after the destructor of the container object.

Constructor and destructor Chaining

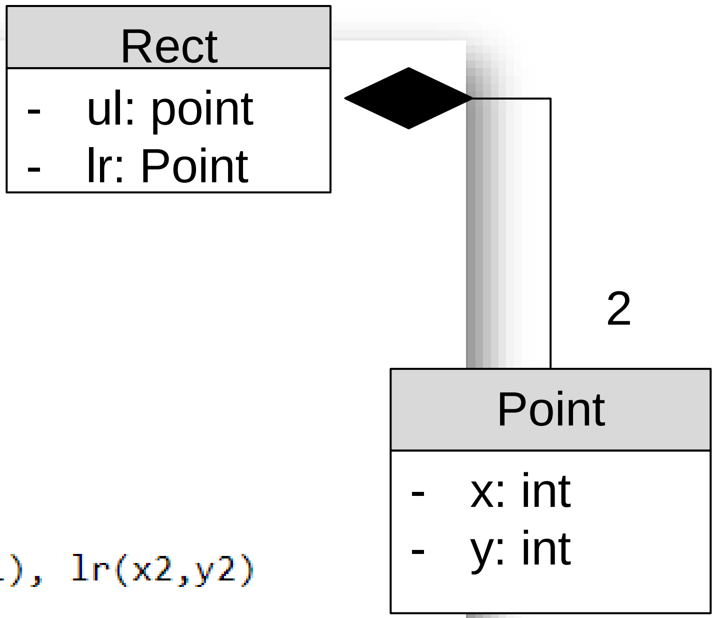
- The question is, which constructor of the embedded object is calling? Where it is not allowed to initialize any attributes inside the class.
- The answer is: the default constructor of the embedded objects which will be called.
- What if there is no default constructor for its class? Or if I want to let another constructor to be called instead of the default constructor.
- Yes we can, by making redirection to another constructor through the header of the constructor of the container object. In the above example, we make the redirection of the Point objects constructors through the header of the Line constructor, by butting ":" at the end of the header and write the name of embedded object with the parameters to select which constructor you want. As we show below:

```
Line(int x1, int y1, int x2, int y2) : start(x1, y1) , end(x2, y2)
{}
```

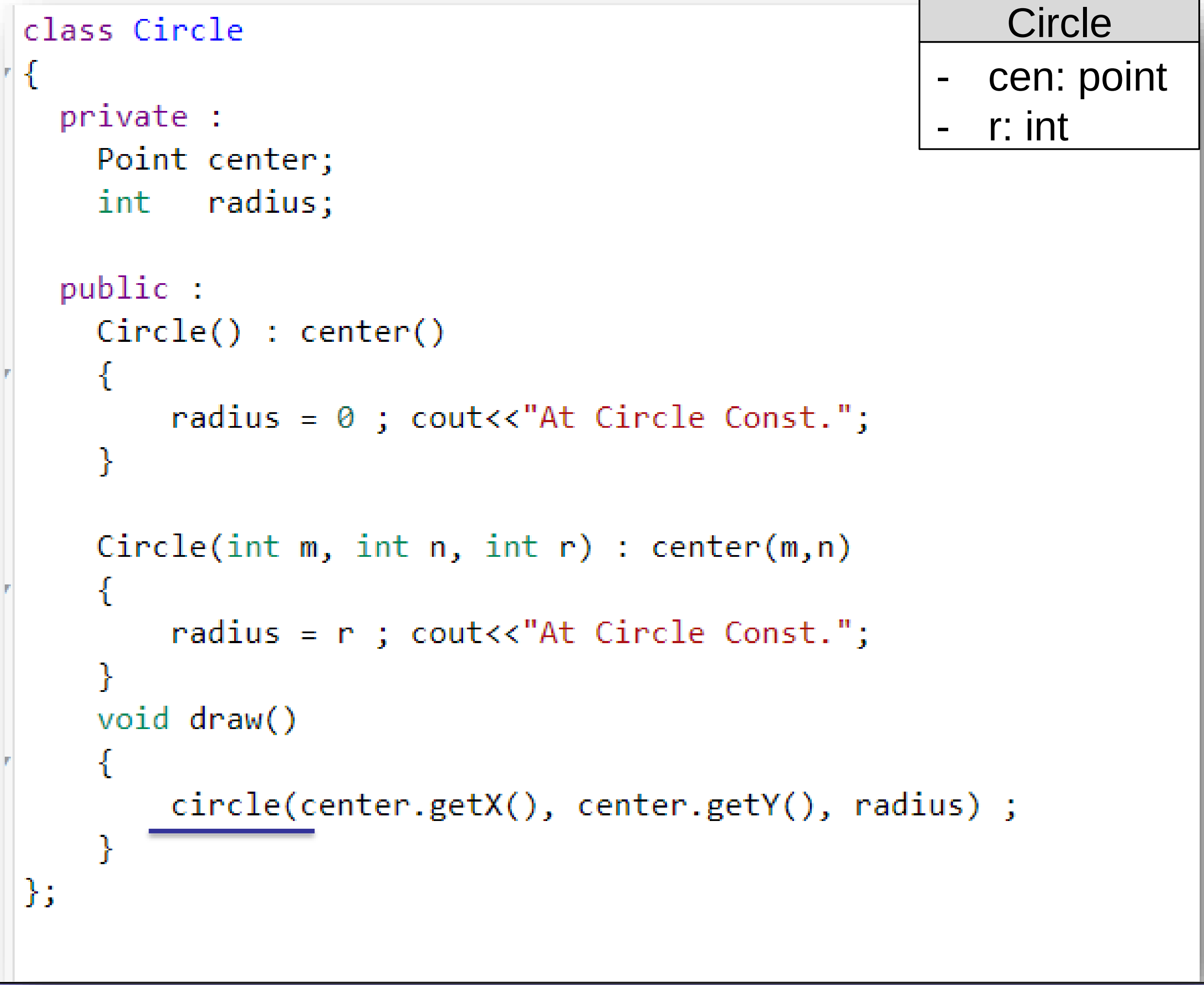
Example



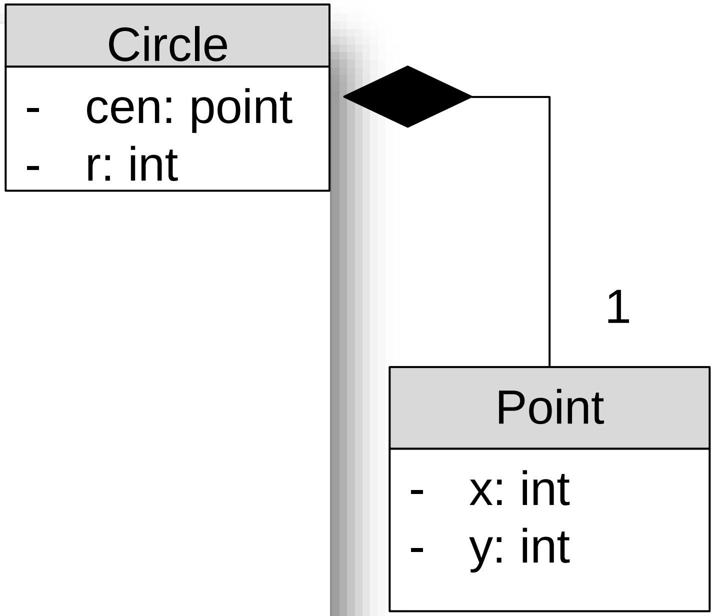
```
class Rect
{
  private:
    Point ul;
    Point lr;

  public:
    Rect() : ul(), lr()
    { cout<<"At Rect Const.";   }

    Rect(int x1, int y1, int x2, int y2) : ul(x1,y1), lr(x2,y2)
    { cout<<"At Rect Const.";   }

    void draw()
    {
        rectangle(ul.getX(), ul.getY(), lr.getX(), lr.getY()) ;
    }
};
```



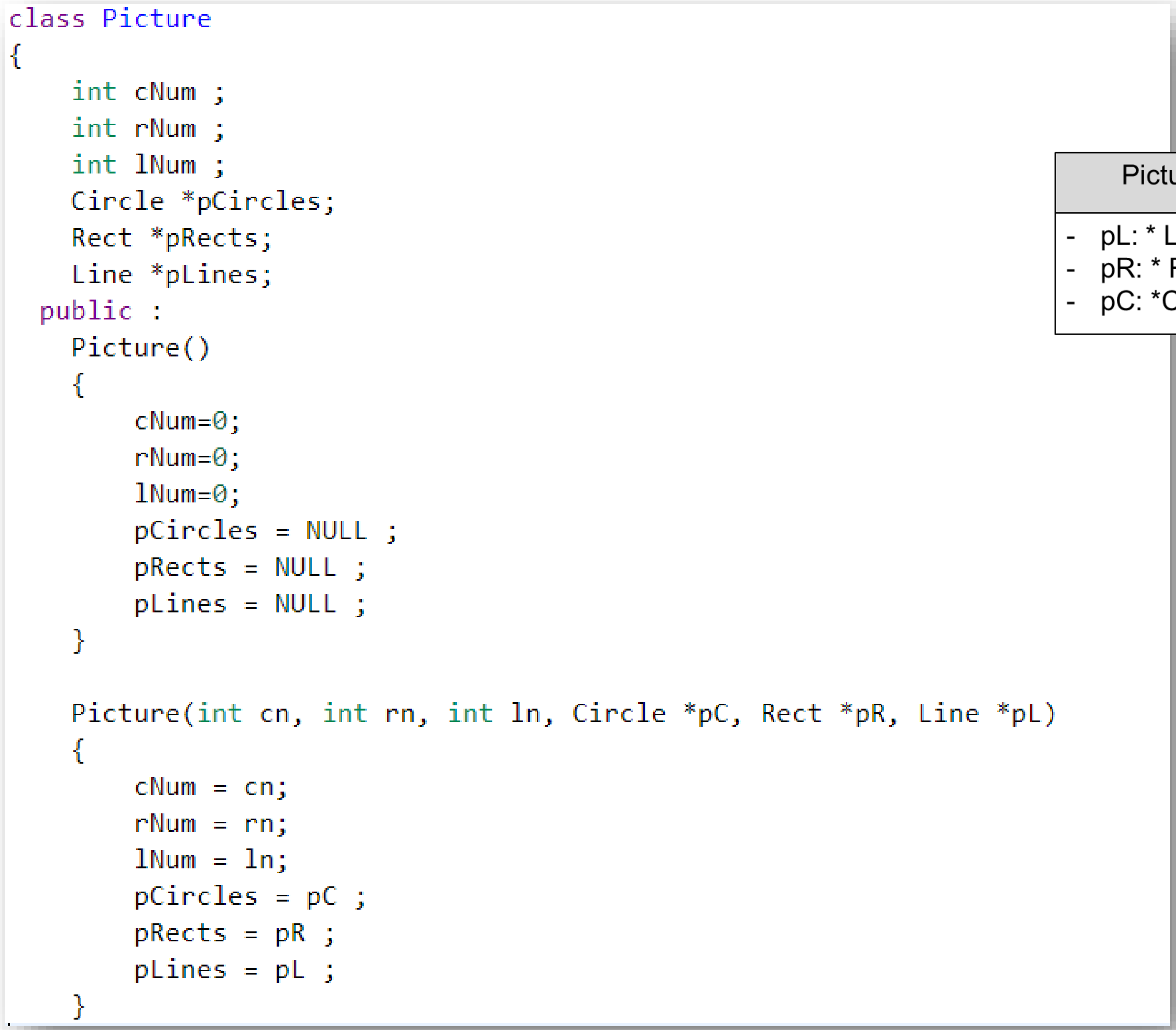
Example



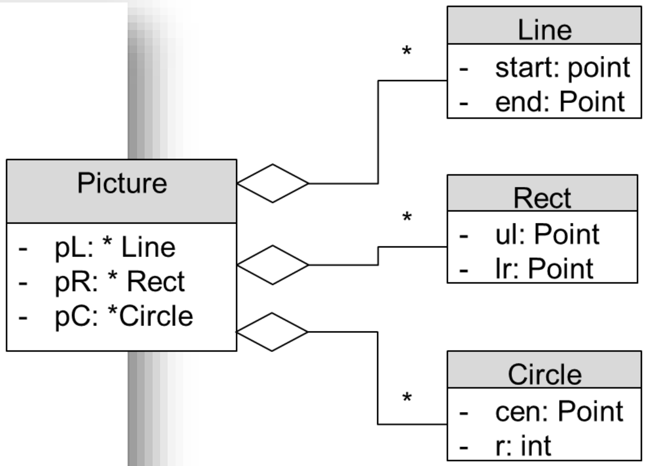
```
class Circle
{
  private :
    Point center;
    int   radius;

  public :
    Circle() : center()
    {
        radius = 0 ; cout<<"At Circle Const.";
    }

    Circle(int m, int n, int r) : center(m,n)
    {
        radius = r ; cout<<"At Circle Const.";
    }
    void draw()
    {
        circle(center.getX(), center.getY(), radius) ;
    }
};
```



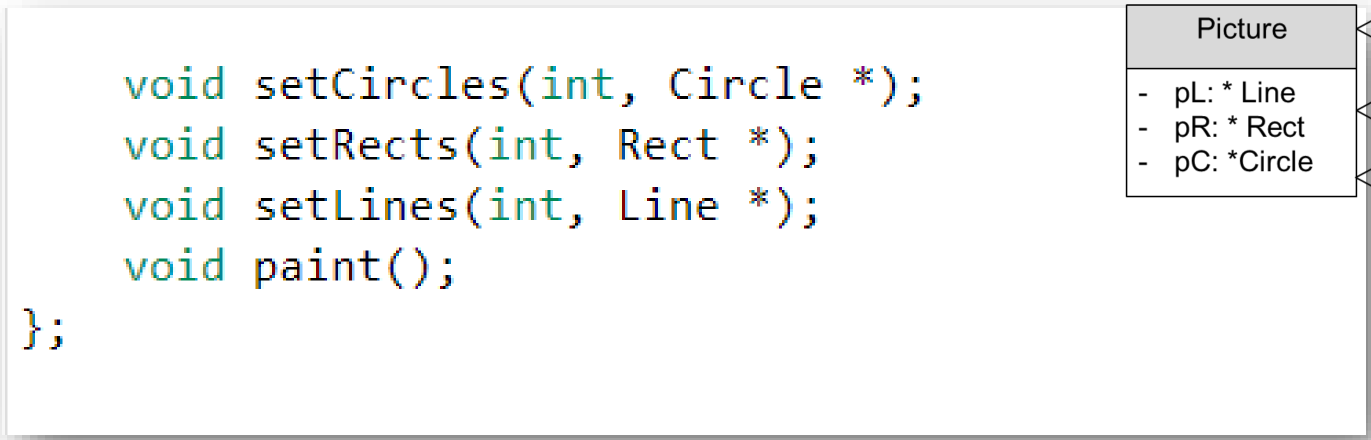
Example



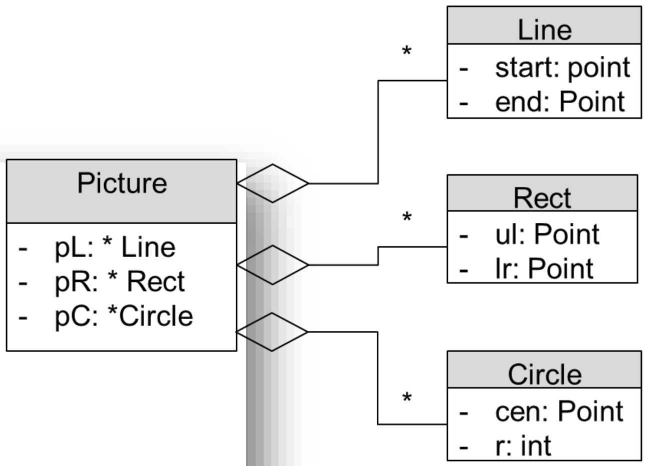
```
class Picture
{
    int cNum ;
    int rNum ;
    int lNum ;
    Circle *pCircles;
    Rect *pRects;
    Line *pLines;
  public :
    Picture()
    {
        cNum=0;
        rNum=0;
        lNum=0;
        pCircles = NULL ;
        pRects = NULL ;
        pLines = NULL ;
    }

    Picture(int cn, int rn, int ln, Circle *pC, Rect *pR, Line *pL)
    {
        cNum = cn;
        rNum = rn;
        lNum = ln;
        pCircles = pC ;
        pRects = pR ;
        pLines = pL ;
    }
```



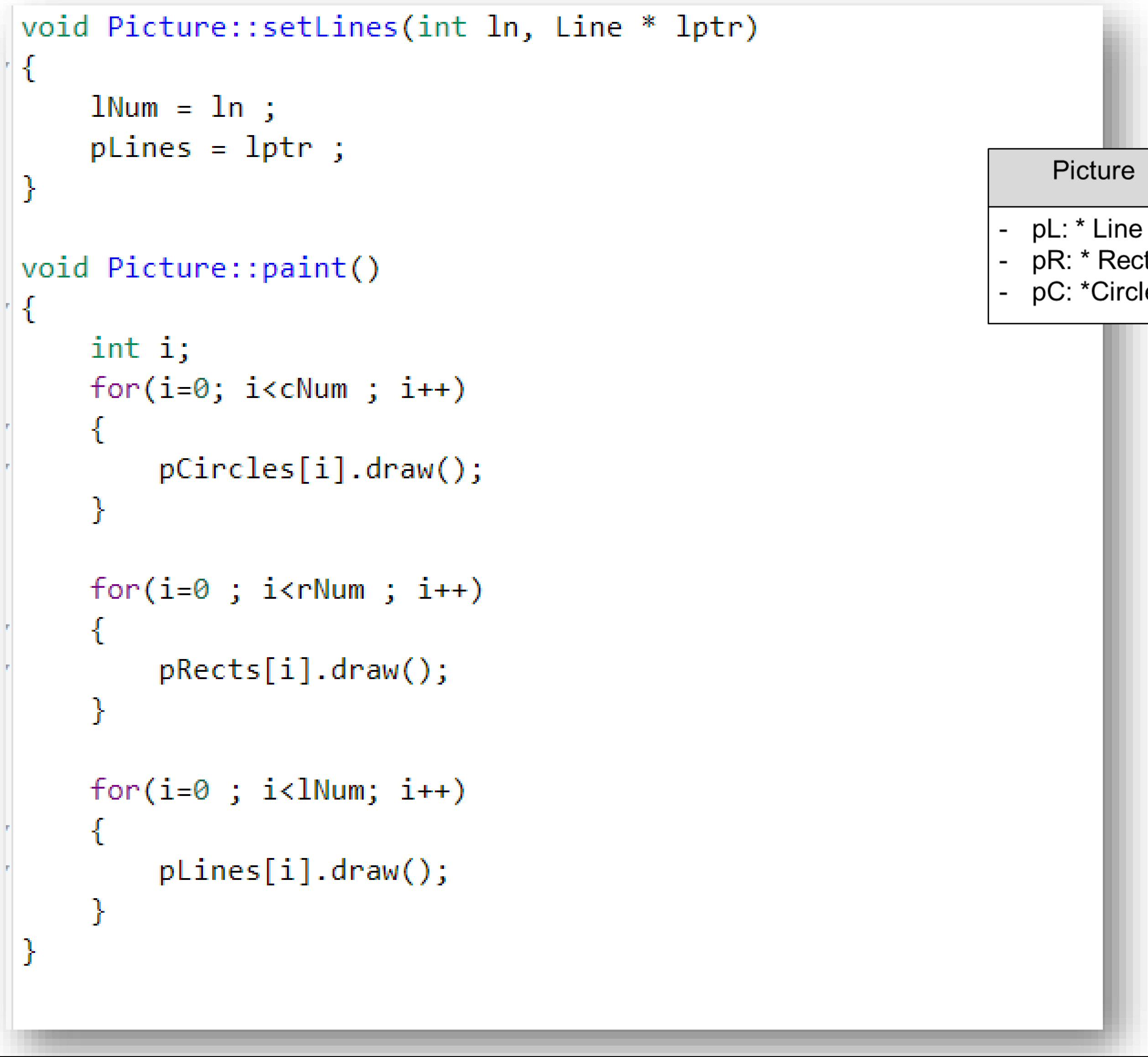
Example



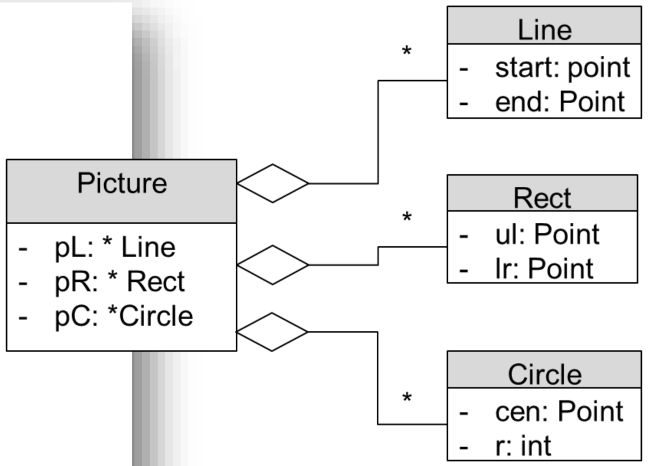
```
    void setCircles(int, Circle *);
    void setRects(int, Rect *);
    void setLines(int, Line *);
    void paint();
};
```



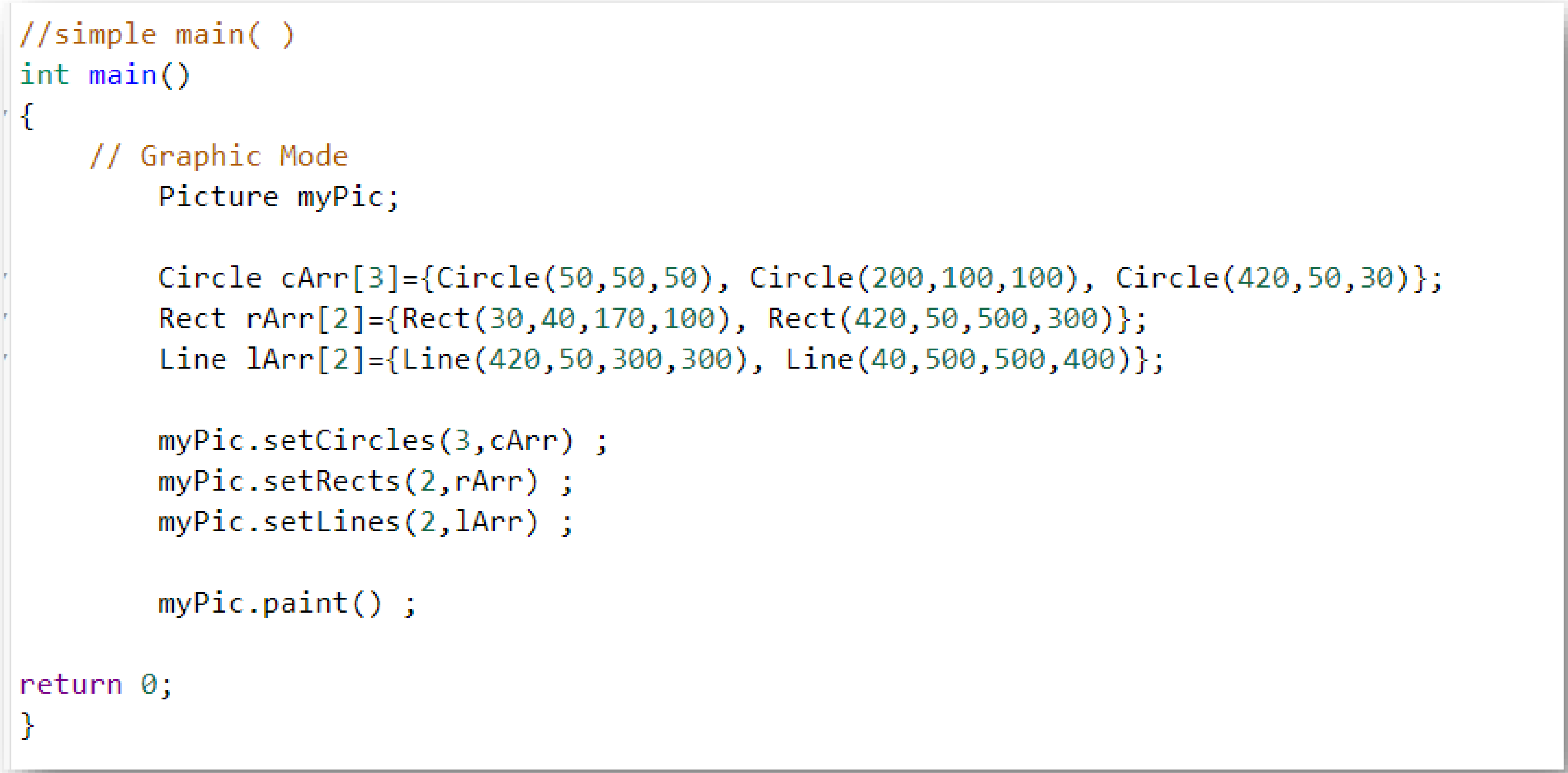
Example



```
void Picture::setLines(int ln, Line * lptr)
{
    lNum = ln ;
    pLines = lptr ;
}

void Picture::paint()
{
    int i;
    for(i=0; i<cNum ; i++)
    {
        pCircles[i].draw();
    }

    for(i=0 ; i<rNum ; i++)
    {
        pRects[i].draw();
    }

    for(i=0 ; i<lNum; i++)
    {
        pLines[i].draw();
    }
}
```



Example



```
//simple main( )
int main()
{
    // Graphic Mode
        Picture myPic;

        Circle cArr[3]={Circle(50,50,50), Circle(200,100,100), Circle(420,50,30)};
        Rect rArr[2]={Rect(30,40,170,100), Rect(420,50,500,300)};
        Line lArr[2]={Line(420,50,300,300), Line(40,500,500,400)};

        myPic.setCircles(3,cArr) ;
        myPic.setRects(2,rArr) ;
        myPic.setLines(2,lArr) ;

        myPic.paint() ;

return 0;
}
```

Example

```
int main()
{		Picture   myPic;
	//example on static allocation
		Circle cArr[3]={Circle(50,50,50), Circle(200,100,100), 
Circle(420,50,30)};
	//example on static allocation, using temporary objects (on the fly)
		Rect rArr[2] ;
		rArr[0] = Rect(30,40,170,100) ;
		Point myP1(420,50) ;
		Point myP2(500,300) ;
		rArr[1] = Rect(myP1, myP2) ;
```

Example

```
int main()
{           Picture   myPic;
        :
        //example on dynamic allocation, using temporary objects (on the fly)
            Line * lArr ;
            lArr = new Line[2] ;
            lArr[0] = Line(Point(420,50) , Point(300,300)) ;
            lArr[1] = Line(40,500,500,400) ;
            myPic.setCircles(3,cArr) ;
            myPic.setRects(2,rArr) ;
            myPic.setLines(2,lArr) ;
            myPic.print() ;
            delete[] lArr ;
```

Chapter 5 Outline

1. Collections of objects
   1. Static Allocations : array of objects
   2. Dynamic Allocation: pointer to object
2. Class Relations
   1. Association
   2. Aggregation
   3. Composition
3. Example

Lab Exercise

Lab Exercise

- Example of Association and Composition :
  - Picture, Point, Rect, Circle, Line

- Collections: Simple trials in the main( ) function to create object and array of objects from classes of (Line , Rect,Circle)

- Note: You should Trace the code.