Object-oriented programming

Lecture #6: Objects and Classes in C++

Outline

- Class/Object Definition
- Member Access
- Member Function
- Object Copy: Deep vs. Shallow

Classes and Their Instances

- *class class_name{ Member_List }; //Class*

  Member_List ::= MemberVariables | MemberFunctions

- *class_name identifier; //Object*

Class Definition

```
class Point {
  private:
    int x,y; //coordinates
  public:
    Point(int xVal = 0, int yVal = 0) { x = xVal; y = yVal; }
    int getX() { return x; }
    int getY() { return y; }
};
```

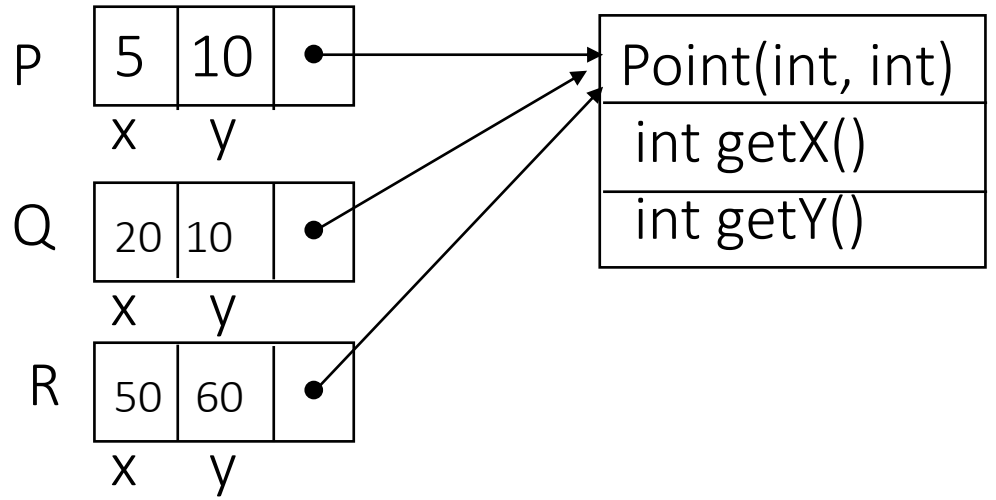
Member Function

- A function that has its definition or its prototype within the class definition
- (Many) used interchangeably with **Methods**

Class Members

- A class object contains a copy of the data defined in the class
  - The functions/methods are shared
  - The data are not

```
Point P(5,10);
Point Q(20,10);
Point R(50,60);
```



Point Objects

//Lec6_ex1-Point.cpp

Controlling Member Access

```
class class_name {
    public:
     //public members
    protected:
     //protected members
     private:
     //private members
};
```

Access Rules

Type of Member	Member of the Same Class	Friend	Member of a Derived Class	Non-Member Function
Private (Default)	X	X		
Protected	X	X	X	
Public	X	X	X	X

Access Rules

- For public member:
  - You can get to the member using the “.” (dot) operator
  - p.getX();
- For the other two member types:
  - NO!

Point (ex2)

```
class Point {
  private:
    int x;
  public:
    int y;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int xVal) { x = xVal; }
    void setY(int yVal) { y = yVal; }
};
```

Point (ex2)

```
Point p;
p.setX(300);
p.setY(500);
cout << “x = ” << p.getX() << endl;
cout << “y = ” << p.y << endl;
//Lec6_ex2-Point.cpp
```

=> What the printed values?

Example about friend class

- Lec6_ex3_FriendClass.cpp

```
#include <iostream>
class A {
    private:
        int a;
    public:
        A() { a=10; }
        void seta(int value);
    friend class B;     // Friend Class
};
void A::seta(int value) { a=value; }
class B {
    private:
        int b;
    public:
        void showA(A& x) { std::cout << "A::a=" << x.a;   }
};
```

```
int main() {
    A a;    B b;
    a.seta(15);
    b.showA(a);
    return 0;
}
```

Inline Functions

- In-line: functions are defined within the body of the class definition.
- Out-of-line: functions are **declared** within the body of the class definition and **defined** outside
- *inline* keyword: Used to define an inline function outside the class definition.

```
class Point {
 ...
 public:
    void setX(int valX) {x=valX;}; // set x
    void setY(int valY);  //set y
    void delta(int, int); //adjust the coordinators
};
 void Point::setY(int valY) { y=valY; }
 inline void Point::delta(int dx, int dy){ x +=dx; y+=dy; }
```

Omitting the name is allowed

In-line

Out-of-line

Also In-line

Constant Member Function

- To guarantees that the state of the current class object is not modified by the function

```
class Point {
 public:
 …
    int getX() const { return x; }
};
```

- Another example: *Lec6_ex4_ConstantMember_Function.cpp*

Constructors

- Called to create and object
- Declared with the name: *classname(….);*
- **Default constructors:** no parameters
- **Alternate constructors:** with parameters

Constructors

- **Overloading Constructors**: with different parameters
- Point() and Point(int, int) are overloaded constructors

Constructors

- **Copy Constructor:** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously
- If a copy constructor is not defined in a class, the compiler itself defines one

Constructors (E.g.)

- Point p; //default
- Point a(p), b = p; //copy
- Point c(20,30); //alternate
- Point figure[3]; //default
- Point figure[2] = {p,c}; //copy

Pointers to Class Objects

- Student *p;
- p = new Student;
- if (!p) //could not create *Student*
- Point * figure = new Point[10];

// call Point() 10 times

Destructors

- Called when an object is to be deleted
- Declared with the name: *~classname();*

(no parameters)

- Example:
  - ~Point() { cout << “Destructor called.”; }

Deep Copy Operation

- Deep copy -- copying the contents of an object

```
char name[] = “John Smith”;
char * cp = new char[30];
strcpy(cp,name);
```

- Shallow copy -- copying the pointer of an object

```
char name[] = “John Smith”;
char * cp = name;
```

Student Copy

```
#include <string>
class Course {
private:
    string name;
public:
    Course() { }
    Course( const string & cname );
};
```

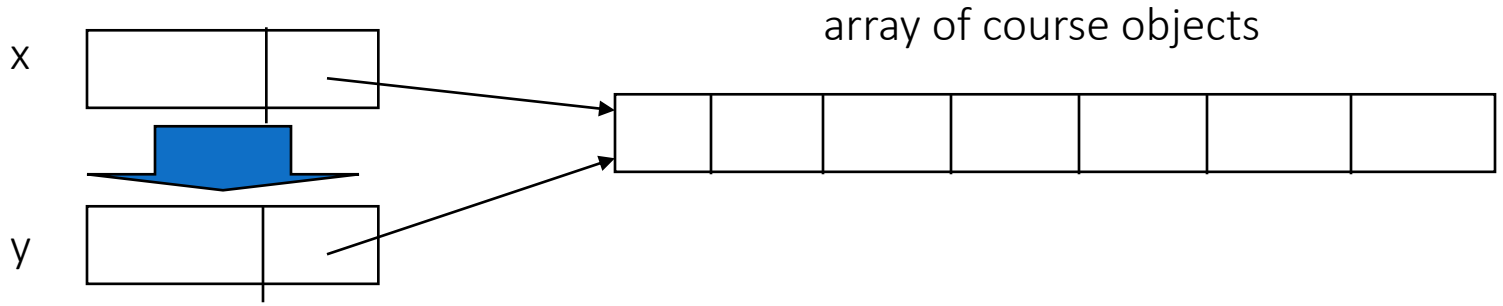
Student Copy

```
class Student {
private:
  Course * coursesTaken;
  unsigned numCourses;
public:
  Student( unsigned nCourses );
  ~Student();
};
```

```
Student::Student( unsigned nCourses ) {
  coursesTaken = new Course[nCourses];
  numCourses = nCourses;
}
Student::~Student() {
  delete [] coursesTaken;
}
```

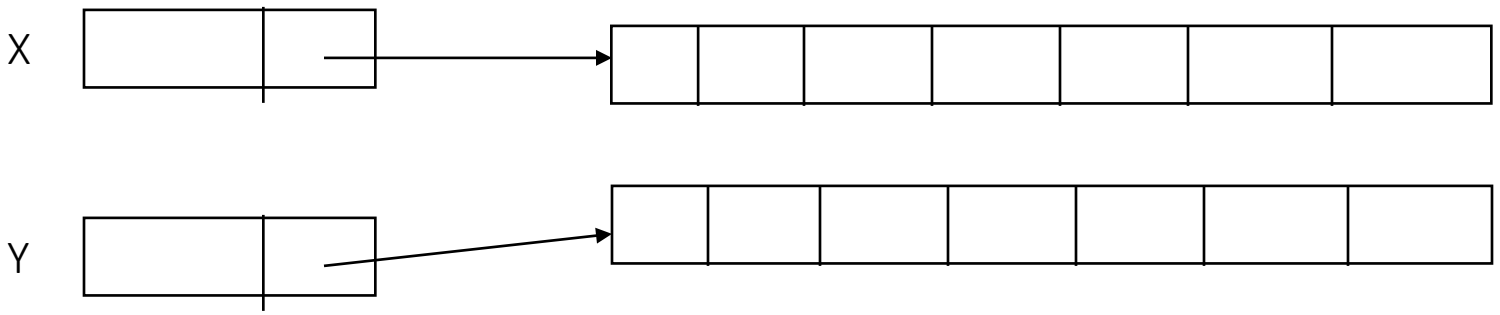
Student Copy

```
int nCourses = 7;
Student x(nCourses);
Student y(x);
```



Student Copy

- If you'd like the following:



Student Copy

```
Student( const Student & s) {
    numCourses = s.getNumCourses();
    courseTaken = new Courses[numCourses];
    for(int i = 0; i < numCourses; i++)
        courseTaken[i] = s.getCourse(i);
}
```

Composite Classes

- A class which contain data members that are objects of other classes
- When a class contains *instances*, *references*, or *pointers* to other classes, we call it a composite class

Composite Classes

- E.g.: A Circle contains a Point; a Figure contains an array of Points

Pre-defined Order

- The constructors for all member objects are executed in the order in which they appear in the class definition. All member constructors are executed before the body of the enclosed class constructor executes.
- Destructors are called in the reverse order of constructors. Therefore, the body of the enclosed class destructor is executed before the destructors of its member objects.

Exercise

- To write a program to test this order.

Exercise