Outline

- Introduction
- Initializing composite objects
- Constant data members: where to initialize
- Composite classes share objects?
- Member initialization list

Objects as members of their own class?
- Reference to objects
- Pointers to objects
- Static objects

- Container class of components
- Nested classes

Composition versus inheritance: a case study “Circle/Point”
Class composition

- Class composition means a class contains an object of a different class
  - A class that has objects of other classes as their data members are called *composite classes*
  - An object member of a class is called *component object*
  - The class of a component object is called *component class*

- Member function of a composite class cannot access private members of included objects

- For example, a class Date would be composed of a Time object
  - → Class Date is a client (or user) of class Time
class Time {
    public:
    void setTime ( int , int , int );
    void printTime ( ) const ;

    private:
    int hour , minute , second ;
};

class Date {
    public:
    Date ( int , int , int , int , int , int , int );
    void printDate ( ) const ;

    private:
    int day , month , year ;
    Time time ;
};
// Think how you would design it better?
Date::Date( int d, int m, int y, int sec, int min, int hour ) {
    day = d;
    month = m;
    year = y;
    time.setTime(sec, min, hour);
}

void Date::printDate() {
    cout << month << '/' << day << '/' << year << ' ';
    time.printTime() ;
}

Can class Date access private members of class Time?
// Listing 12.1. a composite object with wasted constructor calls

class Point {
private:
    int x, y;                 // private coordinates

public:
    Point (int a=0, int b=0)  // general constructor
    {
        x = a;  y = b;
        cout << " Created: x= " << x << " y=" << y << endl;
    }
    ....
};
```cpp
class Rectangle {
    // a composition of points?

private:
    Point pt1, pt2;  // top-left, bottom-right corner points
    int thickness;   // thickness of the rectangle border

public:
    Rectangle (const Point& p1, const Point& p2, int width=1);
    ...
};
```
Testing Point/Rectangle

Point p1(20,40), p2(70,90); // top-left, bottom-right corners
Rectangle rec(p1,p2,4); // construct rec

// alternatively (bad choice)
// class Rectangle // a composition of points?
//{
// private:
// int x1,y1,x2,y2; // top-left, bottom-right corner points
// int thickness; // thickness of the rectangle border
// public:
// Rectangle (const int xx1, const int yy1,
// const int xx2, const int yy2, int width=1);
Initializing composite objects

- All data members could be specified in the member initialization list (MIL)

- However,
  1. For members that are themselves class objects, they should be initialized in the member initialization list
  2. Constant data members (including references) can only be initialized inside the member initialization list

- Object data members are first created before the composite class constructor is called
Initializing composite objects

With wasted constructor calls

Listing 12.1. A composite object with wasted constructor calls

class Point {
  private:
    int x, y; // private coordinates
  public:
    Point (int a=0, int b=0) // general constructor
    {
      x = a; y = b;
      cout << "Created: x= " << x << " y= " << y << endl;
    }
    Point (const Point& pt) // copy constructor
    ...
    void operator= (const Point& pt) // assignment operator
    ...
    .....}

© A. Amer
Initializing composite objects

Wasted calls

class Rectangle {
    private:
    Point pt1, pt2;        // top-left, bottom-right corner points
    int thickness;         // thickness of the rectangle border

    public:
    Rectangle (const Point& p1, const Point& p2, int width=1);
    ....
};

Rectangle :: Rectangle (const Point& p1, const Point& p2, int width) {
    pt1 = p1; pt2 = p2; thickness = width; } // set data members
Initializing composite objects

Wasted calls

Point p1(20,40), p2(70,90);
// 1. p1 and p2 created, Point general constructor called

Point point(100,120);
// 2. point is created, Point general constructor called

Rectangle rec(p1,p2,4);
// 3. Creation of Rectangle object rec
// 3.1 creation of pt1 and pt2, calling Point default constructors
//   Note: Point default constructors are called before the
//   the Rectangle object is created!
// 3.2 assignment operator is called
Initializing composite objects

Wasted calls

- For large composite objects, the process of creation can become quite wasteful. What to do?

- The solution: Member initialization list (MIL)

```cpp
Rectangle :: Rectangle(const Point& p1, const Point& p2, int w)
    : thickness(w), pt1(p1), pt2(p2)  // initialization list
{  }  // empty member body
```
Initializing composite objects

Saving calls

```cpp
// Listing 12.2. A composite object without wasted constructor calls
class Point {
    private:
        int x, y; // private coordinates
    public:
        Point (int a=0, int b=0) // general constructor
        {
            x = a; y = b;
            cout << "Created: x= " << x << " y= " << y << endl;
        }
        Point (const Point& pt) // copy constructor
            ...
        void operator = (const Point& pt) // assignment operator
            ...
    }
};
```
Initializing composite objects

Saving calls

class Rectangle {
    private:
        Point pt1, pt2; // top-left, bottom-right corner points
        int thickness; // thickness of the rectangle border

    public:
        Rectangle (const Point& p1, const Point& p2, int width = 1);
        ...;
};

// Better choice: copy constructor calls
Rectangle::Rectangle(const Point& p1, const Point& p2, int w)
    : thickness(w), pt1(p1), pt2(p2) // initialization list
{ }
    // empty member body
Initializing composite objects

Saving calls

Point p1(20,40), p2(70,90);
// 1. p1 and p2 created, Point general constructor called

Point point(100,120);
// 2. point is created, Point general constructor called

Rectangle rec(p1,p2,4);
// 3. Creation of Rectangle object rec
// 3.1. pt1 and pt2 created, Point copy constructor called
// 3.2 No assignment operator is called! Saving!
Initializing composite objects

Summary

- The purpose of the member initialization list is to avoid a call to the default constructor of the component class before the call to the composite class constructor.

- The member initialization list replaces the call to the default constructor of the component class with the call to the constructor specified in the initialization list.
Outline

- Introduction
- Initializing composite objects
- Constant data members: where to initialize
- Composite classes share objects?
- Member initialization list
  Composition versus inheritance: a case study “Circle/Point”
- Objects as members of their own class?
  - Reference to objects
  - Pointers to objects
  - Static objects
- Container class of components
- Nested classes
Constant variable?

Many usage of "const":

```c
const int x; // constant int
x = 2; // illegal – can’t modify x

const int * pX; // changeable pointer to constant int
*pX = 3; // illegal – can’t use pX to modify an int
pX = &someOtherIntVar; // legal – pX can point somewhere else

int * const pY; // constant pointer to changeable int
*pY = 4; // legal – can use pY to modify an int
pY = &someOtherIntVar; // illegal – can’t make pY point anywhere else

const int * const pZ; // const pointer to const int
*pZ = 5; // illegal – can’t use pZ to modify an int
pZ = &someOtherIntVar; // illegal – can’t make pZ point anywhere else
```
**Constant variable?**

```c
int y;
const int* pConstY = &y; // legal – but can’t use pConstY to modify y
int* pMutableY = &y; // legal – can use pMutableY to modify y
*pMutableY = 42;

const int x; // x cannot be modified

const int* pX = &x; // pX is the address of a const int
// and can’t be used to change an int
*pX = 4; // illegal – can’t use pX to change an int

int* plnt; // address of normal int
plnt = pX; // illegal – cannot convert from const int* to int*

int *plnt; // address of a normal int
plnt = &x; // illegal – cannot convert from const int* to int*
```
Constant data members

Where to initialize?

- What is common in the following data? account number, employee date of hire, price paid, ID, weight of one unit of area

- How often they change? Does it make sense to change them?

- Recall: with const int x=10;
x cannot be assigned any value later

- In analogy: a constant data member has to be initialized immediately after it is created

- How can we do it?
Constant data members

Member initialization list

class Rectangle {
    private:
    Point pt1;
    Point pt2;
    int thickness;
    const double weight;  // weight of one unit of area

    public:
    Rectangle (const Point& p1, const Point& p2, double wt, int width = 1);
    ...;
};

Rectangle::Rectangle (const Point& p1, const Point& p2, double wt, int width)
    : thickness(width), pt1(p1), pt2(p2), weight(wt)
    // initialization of weight(wt) here is mandatory
{ }
Constant data members

Another example

class MyClass {

private:
    int hours; // Is 'hours' declared here?
    int minutes;
    const int size;

public:
    MyClass(int h, int m, int s)
        : hours(h), minutes(m), size(s) // member initialization list
    {
    }
    void print() const
    {
        cout << "your class data are \n";
        cout << " Hours: " << hours << ' ' << "Minutes:"
            << minutes << ' ' << "Size: " << size << endl;
    }
}
Constant data members

```cpp
void set(int a, int b, int c) {
    // size=a; // ***TRY IT****: uncomment this line to test
    // so you want to change size?
    // Well, since size is constant the compiler will not let this go
    // const_test.cpp:56: assignment of read–only member ‘MyClass::size’
    hours = b;
    minutes = c;
}

void operator = (const MyClass &c) {
    hours = c.hours;
    minutes = c.minutes;
    // size = c.size; // ***TRY IT****: uncomment this line to test
    // the compiler will not agree!! size is const => cannot be changed!
    // Error: assignment of read–only member ‘MyClass::size’
}
}; // end class
```
Constant data members

MyClass mc(5,10,60);
// with this you are declaring & initializing
// the object mc including hours, minutes and size.
// They are being created ("constructed") with a specified value.
// Thus they are being initialized. If you change their
// value later with operator = that is an assignment.
// like   mc=mc1;
mc.print();  // prints 5,10,60
mc.set(6,5,7);
mc.print();  // prints 6,5,60

MyClass mc1(15,20,70);
// so you want to assign mc1 to mc
// this implies you would change the const data member size!!!
mc=mc1;  // size is not/cannot be changed!
mc.print();  // prints 15,20,60
Composite classes share objects?

Can a data member of a class be shared by two composite class objects?

An object can be shared using reference or pointer to an object.
Outline

- Introduction
- Initializing composite objects
- Constant data members: where to initialize
- Composite classes share objects?
- Member initialization list
  Composition versus inheritance: a case study “Circle/Point”
- Objects as members of their own class?
  - Reference to objects
  - Pointers to objects
  - Static objects
- Container class of components
- Nested classes
Reference data members

- Recall: all references in C++ are constant:
  1. they cannot change after they are initialized
  2. but their content can change

- Hence,
  reference data members must be initialized only in a member initialization list

- Note:
  - `Point& pt1;`  
    → pt1 is a reference (an alias) that cannot change but its content can be changed
  - `const Point& pt1;`  
    → pt1 is a reference that cannot change; its contents cannot change either
Reference data members: Example

```cpp
class Rectangle {
    private:
        Point& pt1; // shared: reference cannot be changed
        Point& pt2; // shared: reference cannot be changed
        int thickness; // thickness of the rectangle border
        const double weight; // weight of one unit of area

    public:
        Rectangle (const Point& p1, const Point& p2, double wt, int width=1);
        ....
};
```
Reference data members: Example

```cpp
Rectangle :: Rectangle(const Point& p1, const Point& p2, double wt, int width)
    : pt1(p1), pt2(p2), weight(wt)
    // pt1(p1) is mandatory because ref is constant
{ thickness=width; }
```

Q: What about the rightmost parameters in a constructor? Can you tell what is wrong with ...

```cpp
Rectangle (const Point& p1, const Point& p2, int width=1, double wt);
// Hint: Rectangle R(p1, p2, width, wt); or Rectangle R(p1, p2, wt, width);
```
Pointers members of composite classes

class Rectangle {
    private:
        Point *pt1; // can be shared and changed
        Point *pt2;
        int thickness; // thickness of the rectangle border
        const double weight; // weight of one unit of area

    public:
        Rectangle (const Point* p1, const Point* p2, double wt, int width=1);
        ....
};

Rectangle::Rectangle (const Point* p1, const Point* p2, double wt, int width)
    : thickness(width), pt1(p1), pt2(p2), weight(wt)
    // pt1(p1) is pointer initialization — not copy constructor
    // You may pass Pointers as not const; Then they can be changed
{ }

Pointers members of composite classes

Point* p = new Point(20, 40);
Point* q = new Point(70, 90);

Rectangle rec(p, q, 4);
// 3. Creation of Rectangle object rec
// 3.1. pt1 and pt2 pointers are assigned They Point to p and q.
Outline

- Introduction
- Initializing composite objects
- Constant data members: where to initialize
- Composite classes share objects?
- Member initialization list
  Composition versus inheritance: a case study “Circle/Point”
- Objects as members of their own class?
  - Reference to objects
  - Pointers to objects
  - Static objects
- Container class of components
- Nested classes
Member initialization list

Mandatory:
- Constant data members including references (which are constant), constant pointers, ...
- Call to base class constructor
- Reference to a component object

Preferred:
- Initialize component objects
  (i.e., objects as member of other classes)
  it saves extra calls

Optional:
- A pointer to a component object
- Other data members (non-static)
Member initialization list

Practical programmers:

- emphasize initializing ALL data members in ALL constructors (primitive or user-defined)
- initialize most of the data members in the MIL
- initialize data members IN ORDER of declaration unless you ABSOLUTELY cannot (see example below)
Member initialization list

Examples of the “absolutely cannot”

- **When the initialization procedure is complex**

```cpp
    data = new Sample*[MAX_ELEMENT]; // Two dimensional array
    for (unsigned int i = 0; i < MAX_ELEMENT; i++)
        data[i] = new Sample[MAX_ELEMENT];
```

- **When there is an important comment to make before a particular initialization line**

```cpp
    History(): size(3), count(0), idx(0) {
        // The following line is the source of BUG 3876
        data = new Sample[MAX_ELEMENT];
        if (data == NULL) {
            cout<<' 'No memory allocated''<<endl; exit(1);
        }
        cout<<' 'user created default constructor of History called''<<endl;
    }
```
Objects as members of their own class?

Recall:

- Data members are allocated in the order in which they are listed in the class declaration
- Static data (members) are one time allocated at the beginning of program execution
- Static data (members) are destroyed after the program terminate
Objects as members of their own class?

// An objects as member of its own class

class Point {
    private:
        int x, y; // private coordinates
        Point origin; // can I do this?

    ...;
};

Point p; // P is created means
// memory for x and y is allocated
// memory for origin is allocated
// memory for x and y of origin is allocated
// memory for origin of origin is allocated ..
// ... etc. infinite recursive loop!

→ (non-static) objects as members of their own class are disallowed
Reference to objects as members of their own class?

```cpp
// An objects as member of its own class

class Point {
    private:
        int x, y;  // private coordinates
        Point &origin;  // can I do this?
    public:
        Point (Point &org, int a=0, int b=0) : origin(org)
            { x=a; y=b; }  // origin(org) mandatory
            ...
    }
```

© A. Amer
Reference to objects as members of their own class?

Point *p = 0; // p set to null
Point *p = new Point(*p, 80, 90); // allocate memory and set to 80,90
Point p1(p); // *p is used as the origin for p1

→ Do not use (non-static) objects as members of their own class unless you have to
But what is the solution then?
// Listing 12.3 Using static data members and a static member function.

class Point {
    private:
        int x, y;  // private coordinates
        static int count;
        static Point origin;

    public:
        Point (int a=0, int b=0) // general constructor
        {
            x = a; y = b; count++;
            cout << " Created: x=" << x << " y=" << y << " count=" << count << endl;
        }
};
Static objects as members of their own class

```cpp
static int quantity() // const is not allowed
    {
        return count;
    }

~Point() // destructor
    {
        count--; // destructor
        cout << "Point destroyed: x=" << x << " y=" << y << endl;
    }
};
```
static objects as members of their own class

```
int Point::count = 0;       // initialization
Point Point::origin(640,0);  // initialization

// output: 'Created: x=640 y=0 count=1'

cout << " Number of points: " << Point::quantity() << endl;
// output: Number of points: 1

Point p1, p2(30), p3(50,70);
// output: 0,0,2 30,0,3 and 50,70,4

cout << " Number of points: " << p1.quantity() << endl;
// output: number of points: 4
// Output from hidden calls:
// destroyed 50,70 destroyed 30,0 destroyed 0,0
// But why ‘destroyed 640,0’ is not output?
// ==> static data are destroyed after the program terminate
```
A note to return types

- Any difference between:
  1. Polynomial Polynomial::operator = (Polynomial& p);
     //with return *this;
  2. Polynomial& Polynomial::operator = (Polynomial& p);
     //with return *this

There is a BIG difference!

- Without the ’&’ a copy of the object to be returned is made,
  and this copy is subsequently returned

- With the ’&’ you are returning a reference to the object, which
  simply means you are not making a copy of the object to
  return

- There are several implications related to this which we have
  discussed in class
Outline

- Introduction
- Initializing composite objects
- Constant data members: where to initialize
- Composite classes share objects?
- Member initialization list
  - Composition versus inheritance: a case study “Circle/Point”
- Objects as members of their own class?
  - Reference to objects
  - Pointers to objects
  - Static objects
- Container class of components
- Nested classes
A container class includes a ‘dynamically’ created collection of components

Useful: when we need a data type that contains a dynamic collection of values

How often we need such thing?
While the program executes, we would like to add more objects to the container.

Why? for temporary storage or for processing.

Example:
- Container class: customer charge account
- components: customer credit card transactions to be processed
- Processing: totals, taxes, printing, reports, searching, etc.

But why not using an array for that?
With an array we cannot perform high level tasks such as addition, search, validation, setting values of sub components, checking the index, etc.

Container classes are designed to perform these operation for the client code

The client code does not need to deal with low-level implementations
Container classes: an example

- For simplicity: a Sample component class with one data member
- The Sample objects are produced for example by a stock exchange ticker tape, a patient monitoring device, or a pressure observation.
- For simplicity, we assume the values for the Sample class are hard-coded in an array
Container classes: an example

// Listing 12.7 A container class with dynamically-allocated memory

class Sample {   // component class

private:
    double value;  // sample value

public:
    Sample (double x = 0)   // default and conversion constructor
        { value = x; }
    void set (double x)     // modifier method
        { value = x; }
    double get () const     // selector method
        { return value; }
};
class History {  // container class: set value
private:
    int size, count, idx;
    Sample *data;  // could be an array of data

public:
    History() : size(3), count(0), idx(0)  // make array empty
    {
        data = new Sample[size];  // allocate new space
        if (data == NULL)
            { cout << "Out of memory\n"; exit(1); }
    }

~History() { delete [ ] data; }  // free dynamic memory

Note: Since "Sample *data" is not a constant it must not be initialize in the MIL
void add(double); // add a sample at the end

Sample& getComponent() // return reference to Sample
{ return data[idx]; } // can be a message target

void getFirst() { idx = 0; }
bool getNext() { return ++idx < count; }
void average() const; // print average

© A. Amer
void History::add(double s) {
    if (count == size) {
        size = size * 2; // double size if out of space
        Sample *p = new Sample[size];
        if (p == NULL)
            { cout << "Out of memory\n"; exit(1); } // test for success
        for (int i=0; i < count; i++)
            p[i] = data[i]; // copy existing elements
        delete [] data; // delete existing array
        data = p; // replace it with new array
        cout << "new size: " << size << endl; // debugging print
    }
    data[count++].set(s); // use next space available
}
**Class composition: container class**

```cpp
void History::average () const {
    cout << "\n Average value: ";
    double sum = 0;
    for (int i = 0; i < count; i++)
        sum += data[i].get();
    cout << sum/count << endl;
}
```
Testing container class

double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29}; // input data
History h;
for (int i = 0; i < 9; i++)
    h.add(a[i]); // add history
cout << "\nMeasurement history:" << endl << endl;
h.getFirst(); // work is pushed up
do {
    cout << " " << h.getComponent().get();
} while (h.getNext());
h.average();

// Output:
new size: 6
new size: 12
Measurement history: 3 5 7 11 13 17 19 23 29
Average: 14.1111
Container classes

- Container classes with more complex design include sorting, search, removal, insertion, update, comparison
- A significant number of container classes are available from the Standard Template Library
Nested classes

- In C++, we can define a server class inside a client class.
- This way the nested class name is not visible outside of the composite class.
- Nested class definitions can appear in either the private or public section.
- In both cases, Nested class is hidden from the rest of the program.
- Nested class cannot be used to declare variables of its type in other scopes outside the composite class.
Nested classes

class History {
private:
    class Sample {
        // component class
        private:
            double value;       // sample value
        public:
            Sample (double x = 0) // default and conversion constructor
                { value = x; }
            void set (double x)       // modifier method
                { value = x; }
            double get () const       // selector method
                { return value; }
    };

    int size, count, idx;
    Sample *data;

public:
    History() : size(3), count(0), idx(0) { // make array empty
        data = new Sample[size];              // allocate new space
        if (data == NULL)
            { cout << "Out of memory\n"; exit(1); } 
    }
    ....
}

// Use
double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 }; // input data
    History h;
    for (int i=0; i < 9; i++)
        h.add(a[i]);       // add history
...
    Sample s=5; // OK?
    History::Sample s=5; // OK? when?
Outline

- Introduction
- Initializing composite objects
- Constant data members: where to initialize
- Composite classes share objects?
- Member initialization list
  Composition versus inheritance: a case study “Circle/Point”
- Objects as members of their own class?
  - Reference to objects
  - Pointers to objects
  - Static objects
- Container class of components
- Nested classes
Composition or inheritance?

- Composition: the existence of a "has-a" relationship aggregate objects in a class
- Inheritance: the existence of a “is-a” relationship conceptually connect classes in an inheritance hierarchy
- In most problems, inheritance is very useful and should be often considered, however,
  - Inheritance forces the client programmer to learn a considerable amount about the server design, specially when the hierarchy is tall
  - Dependencies between classes in inheritance are very strong
Composition or inheritance?

- Composition is conceptually simpler & links between classes are not as strong
- Composition is, in general, easier to maintain
- Choice depends on degree of similarity between related classes
- If the number of common methods is small & the number of methods to be is large, composition is a good alternative
- If a “is-a” relationship is not clear then use composition:
  - A Rectangle "has a" Point but it is not a Point;
  - A Cylinder "has a" Circle or "is a" Circle?
  - An Address "is a" Name or "has a" Name?
Circle/Cylinder: composition

// Listing 14.2 code reuse through class composition
class Circle {   // original code for reuse
protected:     // inheritance is one of the options
    double radius;  // internal data
public:
    static const double PI;  // it must be initialized
    Circle (double r);  // conversion constructor
    double getLength () const;  // compute circumference
    double getArea () const;  // compute area
    double getRadius () const;
    void set(double r);  // change size
};
const double Circle::PI = 3.1415926536;
class Cylinder { // new class Cylinder
    protected:
    Circle c; // no PI, no radius
    double height; // new code
    public:
    Cylinder (double r, double h) : c(r) // initializer list (no PI) 
        { height = h; }
    double getLength () const { return c.getLength(); } // from class Circle
    double getRadius () const { return c.getRadius(); } // from class Circle
    void set(double r) { c.set(r); } // from class Circle
    double getVolume () const // no getArea()
    {
        double radius = c.getRadius();
        return Circle::PI * radius * radius * height;
    }
};
```cpp
int main() {
    Cylinder cyl1(2.5, 6.0), cyl2(5.0, 7.5); // initialize data
    double length = cyl1.getLength(); // similar to Circle
    cyl1.set(3.0);
    double diam = 2 * cyl1.getRadius(); // no call to getArea()
    double vol = cyl2.getVolume(); // not in Circle
    cout << "Circumference of first cylinder: " << length << endl;
    cout << "Volume of the second cylinder: " << vol << endl;
    cout << "Diameter of the first cylinder: " << diam << endl;
    return 0;
}
```
Circle/Cylinder: inheritance

// Listing 14.3 code reuse through inheritance
class Circle {
    // original code for reuse
    protected:
        // inheritance is one of the options
        double radius;  // internal data

    public:
        static const double PI;  // it must be initialized
        Circle (double r);  // conversion constructor
        double getLength () const  // compute circumference
            { return 2 * PI * radius; }
        double getArea () const;  // compute area
        double getRadius () const;
        void set(double r);  // change size
};

const double Circle::PI = 3.1415926536;
class Cylinder : public Circle { // new class Cylinder
    protected:
    double height; // other data is in Circle
    public:
    Cylinder (double r, double h) : Circle(r) // initializer list (no PI)
    { height = h; } // new code
    double getVolume() const // no getArea()
    { return height * getArea(); } // additional capability
} ;
```cpp
int main() {
    Cylinder cyl1(2.5, 6.0), cyl2(5.0, 7.5); // initialize data
    double length = cyl1.getLength();       // similar to Circle
    cyl1.set(3.0);
    double diam = 2 * cyl1.getRadius();     // no call to getArea()
    double vol = cyl2.getVolume();          // not in Circle
    cout << "Circumference of first cylinder: " << length << endl;
    cout << "Volume of the second cylinder:   " << vol << endl;
    cout << "Diameter of the first cylinder:   " << diam << endl;
    return 0;
}
```
Review

class Rectangle {
  private:
  Point p;
  ...
}
  // which Constructor
Rectangle(Point& q, ...) { p=q; ...} // = default constr. +assignment op
  // or
Rectangle(Point& q, ...) : p(q) { ...} // copy constructor

  // What is in a member initialization list?