C8180 - Lecture 4

Note Title 8/28/2009

<u>Announcements:</u>

- HW due Friday by start of class

Additional functions (Not in class)

```
// Free-standing operator definitions, outside the formal Point class definition
Point operator*(double factor, Point p) {
  return p * factor;                    // invoke existing form with Point as left operand
}

ostream& operator<<(ostream& out, Point p) {
  out << "<" << p.getX( ) << "," << p.getY( ) << ">";       // display using form <x,y>
  return out;
}
```

← in a few slides..

<x, y>

Why outside of class?

C++ does not allow. right operator to be instance of an object!

cout << mypoint; ← ostream makes sure to allow file output also

Inheritance

What is it?

parent class w/ child class(es)

<u>Ex:</u> Rectangle → Square

height — size

width

Child class inherits <u>all</u> attributes and methods

Example: Square class

```
class Square : public Rectangle {
  public:
    Square(double size=10, Point center=Point( )) :
      Rectangle(size, size, center)         // parent constructor
      { }

    void setHeight(double h) { setSize(h); }
    void setWidth(double w) { setSize(w); }

    void setSize(double size) {
      Rectangle::setWidth(size);        // make sure to invoke PARENT version
      Rectangle::setHeight(size);       // make sure to invoke PARENT version
    }

    double getSize( ) const { return getWidth( ); }
};  // end of Square
```

Other issues:

A new type of data:

- We have seen public & private. Public is inherited and private is not.

But what about data which should be private, but also should be inherited?

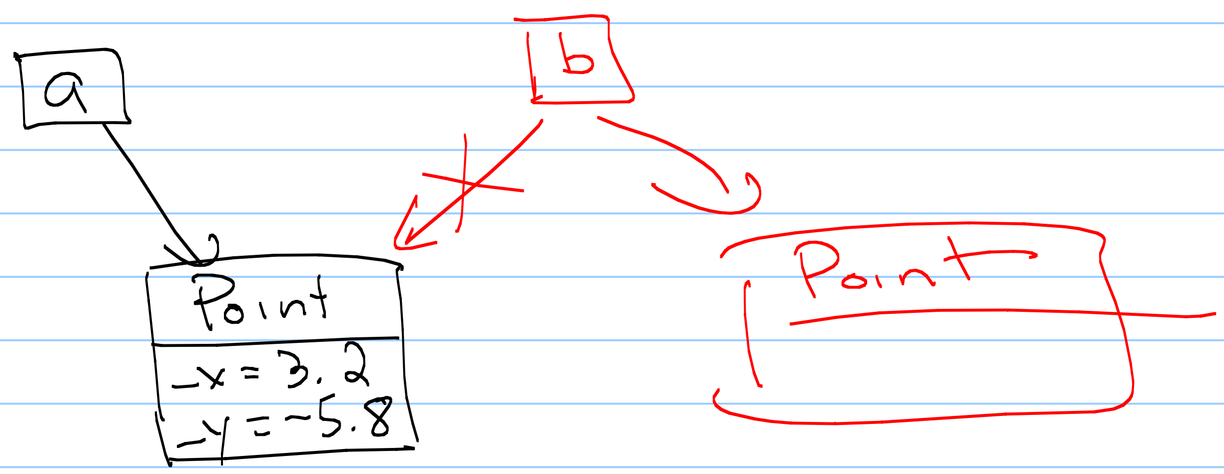
Ex: public: ~~(crossed out)~~

```
public:
    int height;
    int width;
```

```
protected:
    int height;
    int width;
```

Objects & Memory Management

In Python, variables were pointers to data:



What does b = a do?

b = Point(3,4)

C++: A more complicated setup...

C++ allows 3 different models for storing & passing information.

1. Value
2. Reference
3. Pointer

(Remember that strange & a few slides ago?)

Value Variables

When a variable is created, a precise amount of memory is set aside:

```
Point a;
Point b(5, 7);
```

a : Point
x = 0.0 y = 0.0

b : Point
x = 5.0 y = 7.0

This is more efficient, both for space and speed.

Now suppose we set a = b:

a : Point
x = ~~5.0~~ 6
y = ~~7.0~~ 8

b : Point
x = 5.0
y = 7.0

```
Point c(1,1);
a = a + c;
```

They stay separate!

Different than Python:

If I change a, I'd also change b.

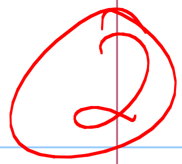
Functions: Passing by Value

```
bool isOrigin(Point pt) {
  return pt.getX( ) == 0 && pt.getY( ) == 0;
}
```

When someone calls isOrigin(myPoint) later, the value pt in the function is initalized as though a <u>new</u> variable was created:

```
Point pt(myPoint);
```

So changes in function to pt don't affect myPoint!



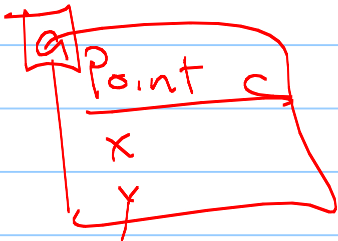
Reference Variables

Syntax:

```
Point& c(a);  // reference variable
```

- c is created as an alias for a
- More like Python model, but can't be changed later

Ex: c=b;
Will not rebind c to point to b, but will change the value of c (and a).



Passing by reference:

Reference variables aren't usually needed in main program.

Instead, they are primarily used for passing to functions.

Ex:

```
bool isOrigin(Point& pt) {
  return pt.getX( ) == 0 && pt.getY( ) == 0;
}
```

↓

Point& pt(myPoint);

in main:

isOrigin(myPoint);

Passing by reference (cont.)

<u>Why?</u>

- faster
- more memory efficient
- any changes inside the function <u>do</u> persist after the fcn

If we want the speed of passing by reference but don't want our object mutated, use const.

```
bool isOrigin(const Point& pt) {
  return pt.getX( ) == 0 && pt.getY( ) == 0;
}
```

If next to input parameter - Compiler will ensure that pt isn't modified.

Speeding up the Point class:

original:

```
double distance(Point other) const {
```

faster:

```
double distance(const Point& other) const {
```

Another:

```
Point operator+(const Point& other) const {
  return Point(_x + other._x, _y + other._y);
}
```

Note: can't make return type Point&

```
Point& operator+(const Point& other) const {
    Point& returnPoint(_x + other._x, _y + other._y)
    return returnPoint;
}
```

Recall: Point output

```
ostream& operator<<(ostream& out, Point p) {
  out << "<" << p.getX( ) << "," << p.getY( ) << ">";     // display using form <x,y>
  return out;
}
```

Here, & is required because streams cannot be copied.

Note that we don't use const, since we are changing the stream by adding data.

Return type: ostream&, which existed in main already