Data Structures

Today:
Classes
Variable Models
Announcements

- New office hours: 1-2pm on Friday (Wed. goes away although I'm often in)

- Lab: due Friday (via git!) (make sure you pass judge program)

- Next HW: half written, half programming - up later today
class Point {
    private:
    double x; // explicit declaration of data members
    double y;

    public:
    Point() : x(0), y(0) { } // constructor

double getX() const {
        return x;
}

void setX(double val) { // mutator
    x = val;
}

double getY() const { // accessor
    return y;
}

void setY(double val) { // mutator
    y = val;
}

}; // end of Point class (semicolon is required)

Figure 9: Implementation of a simple Point class.
Classes:

1. Data & fns: MUST be public, private, or protected (more later)
   - Enforced by compiler!
   - General convention: all data is private

2. Constructor:
   - name: Same as class (only 2 capitalized things)
   - no return type (only timestep)
   - can initialize in list or in body:

Point(double initialX, double initialY) {  
  x = initialX;  y = initialY;  }
More:

3. **No `self`!**

Just say `x` or `y` in class functions and will use class variables.

Note: can't use `x` or `y` as a class variable.

4. Accessor vs. mutator:

   use `const`...

A more complex one...
```cpp
class Point {
private:
    double _x;
    double _y;
public:
    Point(double initialX=0.0, double initialY=0.0) : _x(initialX), _y(initialY) {} // same as simple Point class
    double getX() const { return _x; }
    void setX(double val) {_x = val; } // same as simple Point class
    double getY() const { return _y; }
    void setY(double val) {_y = val; } // same as simple Point class
    void scale(double factor) {
        _x *= factor;
        _y *= factor;
    }
    double distance(Point other) const {
        double dx = _x - other._x;
        double dy = _y - other._y;
        return sqrt(dx * dx + dy * dy); // sqrt imported from cmath library
    }
    void normalize() {
        double mag = distance(Point()); // measure distance to the origin
        if (mag > 0)
            scale(1/mag);
    }
    Point operator+(Point other) const {
        return Point(_x + other._x, _y + other._y);
    }
    Point operator*(double factor) const {
        return Point(_x * factor, _y * factor);
    }
    double operator*(Point other) const {
        return _x * other._x + _y * other._y;
    }
}; // end of Point class (semicolon is required)
```
Notes:

1) \( x + \text{other}.x \):
   allowed only inside class, for when another object is an input.

2) \( \text{operator} + \):
   \( x + y \)

3) two versions of \( \text{operator} \ast \)
Additional common functions, but after class:

```cpp
3. // end of Point 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;
}
```

Why? so we can call

6 * (2,3)

`cout << mypt;`<end>`

"<2,4>"
Finally:

Oh vs. .cpp files:

So far, just used .cpp.
The .oh extension is just for classes.

Idea:

- Separate classes from main, which might need many of them.
- Then import all needed .oh files into one .cpp file that has the main
Inheritance

What is it?

Class is a "subset" of another - can steal all attrs & data

Ex:

Any of graphics objects in Python

Person

/  
faculty  staff  student

  /  
/alum
Code example:

Suppose we make a Rectangle class:
- two private variables: height & width
- functions to reset each

Square class:

```cpp
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
```
And protected data:

- Public
- Private:
- Protected:

Let's for inheritance (a friend class)

Not public but only children + friend classes can see it.
More on variables

In Python, variables were just identifiers for some underlying object. This had implications when passing variables to functions:

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

So if you do:

```python
if (isOrigin(bldg))
```

Figure 14: An example of parameter passing in Python.

...in lists - meant shallow copies
C++: Much more versatile. 3 parameter types

1. Value
2. Reference
3. Pointer

So far, you've been using value - easiest.

Reference & Pointer require looking at memory more carefully...
Value Variables

When a variable is created, a precise amount of memory is allocated:

Point a;
Point b(5,7);

Memory: | labels | content | addresses (hex #s)
--------|--------|---------|-------------------
        |        |         | 867
        |        |         | 868
        |        |         | 869
        |        |         | 870
        |        |         | 871
        |        |         | 872
        |        |         | 873
        |        |         | ... 
        |        |         | 1011
        |        |         | 1012
        |        |         | 1014
        |        |         | 1015
        |        |         | ...
Now:
\[ a = b \]

What happens?
Functions + passing by value:

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

When someone calls `isOrigin(mypoint);`:

The (local) variable `pt` is created as a new, separate variable.

Essentially, compiler inserts `Point pt(mypoint);` as first line of the function.

So—what if we change `pt`?
2. Reference variables

Syntax:

Point & c(a);

What it does: