More with structs...

Objective today is to get more practice with:

- heirarchies of structs
- design and implementation using structs
- assigning structs to structs
- structs as return values

Assigning structs to structs

- taking our points in random order :)
- you can assign structs to each other if they are of the same type
- uses = field by field on the values

```
struct SomeItem {
   float f;
   string s;
};

SomeItem x = { 1.2, “foo” };
SomeItem y;
y = x;
```

```
// the y = x acts the same as
y.f = x.f;
y.s = x.s;
```

Risk of using = on structs

- this only works if = works for each of the field data types
- doesn't copy array fields, because = doesn't work to assign arrays

```
struct ItemWithArray {
   int arr[20];
   float f;
   string s;
};

ItemWithArray a, b;
a = b;  // does copy fields f and s ok
        // does NOT copy the array content
```

Structs as return values

- you can return a struct from a function (a common way of packaging multiple values into a return)
- acts like assigning struct at point of return (with the same risks if the returned struct contains things like arrays)

```
struct SomeItem {
   string str;
   int num;
};
```

```
SomeItem getAnItem()
{
    SomeItem x;
    cin >> x.str;
    cin >> x.num;
    return x
}
```

```
int main()
{
   // called like
   SomeItem myItem = getAnItem();
```

Practice problem: colliding circles

- common problem in games or simulations: given a bunch of shapes in 2d or 3d space, determine which shapes collide with each other/when
- we'll keep it simple and just deal with stationary circles in 2d space: how can we model them and tell which ones overlap?
- possible way to model a circle is as a point (marking its centre) plus its radius ... if we can model a point
- possible way to model a point is as an x,y coordinate pair

Structs for points and circles

```
struct Point {
   float x;
   float y;
};

void fill(Point &pt) {
    cout << “Enter x and y: “;
    cin >> pt.x >> pt.y;
}

void print(Point pt) {
   cout << “(“ << x << “,”;
   cout << y << “)”;
}
```

```
struct Circle {
   Point p;
   float rad;
};

void fill(Circle &c) {
    fill(c.p);
    cout << “Enter radius: “;
    cin >> c.rad;
}

void print(Circle c) {
      print(c.p);
      cout << “:” << c.rad;
}
```

Detecting all collisions

- assume we can write a function to check if two circles overlap

```
int main()
{
   // get our collection of circles
   const int NumCircs = 10;
   Circle circs[NumCircs];
   for (int c = 0; c < NumCircs; c++) {
       fill(circs[i]);
   }
   // in collection, check each circle against
   // all the “later” circles in the array
   for (int first = 0; first < NumCircs-1; first++) {
       for (int sec = first+1; sec < NumCircs, sec++) {
           if (collides(circs[first], circs[sec])) {
              // display info about detected collision
              cout << “collision detected between “;
              print(circs[first]);
              cout << “ and “;
              print(circs[sec]);
              cout << endl;
           }
       }
   }
}
```

Detecting one collision

- two circles collide (overlap) if the distance between their centres is less than the radius of the first plus the radius of the second
- let's assume we can write a function to compute distance between their centres

```
bool collides(Circle c1, Circle c2)
{
    float distance = distBetween(c1.p, c2.p);
    if (distance < (c1.rad + c2.rad)) {
      // they're too close, they overlap
      return true;
    }
    return false;  // didn't overlap
}
```

Getting distance between centres

- formula to compute distance between two points, (x1,y1) and (x2,y2) is well known:

$$(x1-x2)^2 + (y1-y2)^2 = dist^2$$

```
float distBetween(Point p1, Point p2)
{
    float xpart = p1.x - p2.x;
    float ypart = p1.y - p2.y;
    distsq = (xpart * xpart) + (ypart * ypart);
    return sqrt(distsq);
}
```

Gives us all the parts of our program!
Lots of ways to improve efficiency, but that's for another day :)