Constructors and Destructors

`point` class

Let's start with a simple, object-oriented model of a point in two-dimensional space:

```
class point {

   public:
      point(double x, double y);

      double getX() const { return x; }
      double getY() const { return y; }
      void translate(double deltax, double deltay);

   private:
      double x;
      double y;
};
```

I'm hoping there's very little mystery. The **`point`** class is a glorified **`struct`** that encapsulates the data and provides access to that data via **`public`** methods. There's a simple constructor that mandates **`x`** and **`y`** values be specified the instant a **`point`** is declared. And **`translate`** is the only method in place to change a **`point`** once it's been created.

If you learned C++ here at Stanford, it's likely some of the above is confusing. I'm guessing there are your question:

- What does **`const`** mean?
- Why are the implementations of **`getX`** and **`getY`** exposed in the class definition? Aren't we exposing some secret that supposed to be hidden away in some **`.cc`** file?

`const` is a big thing in C++, but it's complicated enough that we avoid it in the CS106 curriculum. When the **`const`** keyword is typed after the signature of a method, it's taken to mean that the method won't change the receiving object during the course of its execution. Restated, the **`const`** marks the receiving object—the object that's addressed by the **`this`** pointer—as immutable.

Note that **`getX`** and **`getY`** are each **`const`** methods, but translate isn't. **`getX`** and **`getY`** don't need to change **`x`** and **`y`** fields in order to read them, and that's why there's **`const`** methods. **`translate`** does change the receiving object, though: It shifts **`x`** and **`y`** values by the specified deltas.

What about the exposed implementations? That's normally a no-no, but when the implementations are super short and super duper obvious, most C++ programmers opt to inline the implementations of trivial one-liner methods right there in the .h file. Yes, you're technically exposing trade secrets, but are they **really** trade secrets when the implementation couldn't be anything else? [Answer: no] At the very least, inlining simplifies the development process and allows you to implement trivial methods without lots of .cc boilerplate. And because the inlined code is exposed during compilation, compilers are able to optimize and reduce the overhead associated with calling and returning from **getX** and **getY**.

The constructor and the translate method are complex enough—complex meaning more than one line—that we type them up as we normally would within a .cc file:

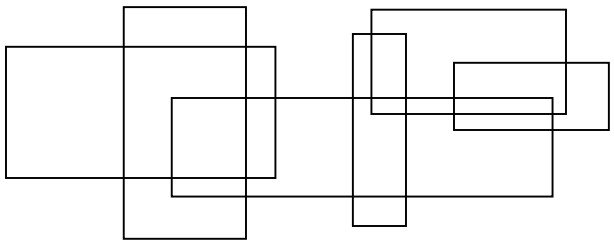
```
point::point(double x, double y)
{
   this->x = x;
   this->y = y;
}

void point::translate(double deltax, double deltay)
{
   x += deltax;
   y += deltay;
}
```

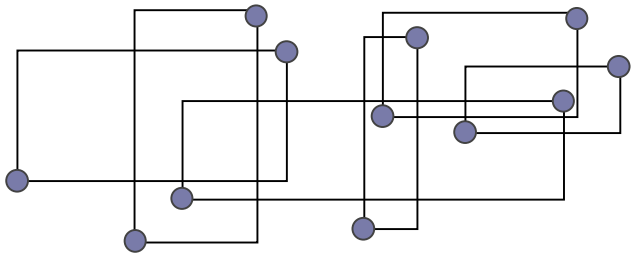
This is all standard fare, and you know precisely what's going on here. The only feature I'm not sure you've all seen before if the use of the this point to disambiguate between **x** the data field and **x** the parameter. Whenever a local variable and an object's data field share the same name, all references to that name are references to the local variable. [You'll hear programming language ~~snobs~~ enthusiasts say that the local variable **x** **shadows** the object data field called **x**.]

Rectangles

Now let's model rectangles—but only those that are aligned with the x and y axes. For those who've not seen rectangles in a while, here are a few:



There are several reasonable ways to model a rectangle, but we'll pretend the only sensible approach is the track a rectangle's lower left and upper right corners. Since the sides of all rectangles are normal to the x and y axes, we know what the rectangle must look like:



Here's the class definition:

```
class rectangle {

   public:
      rectangle(double llx, double lly, double width, double height);

      double getArea() const { return area; }
      void translate(double deltax, double deltay);

   private:
      point ll;
      point ur;
      double area;
};
```

The implementation of **`getArea`** is painfully obvious, so that's why it's inlined (the constructor will initialize the area field—more on that in a second). And the implementation of translate is also pretty simple:

```
void rectangle::translate(double deltax, double deltay)
{
   ll.translate(deltax, deltay);
   ur.translate(deltax, deltay);
}
```

The most interesting thing about this rectangle class is its constructor. Incoming CS107 students could be convinced the following implementation would work:

```
rectangle::rectangle(double llx, double lly, double width, double height)
{
   ll = point(llx, lly);
   ur = point(llx + width, lly + height);
   area = width * height;
}
```

Let's keep talking as if this is all correct. The first line constructs an anonymous point constant out of **`llx`** and **`lly`** so that **`ll`** can be initialized. The second line does that

same thing on behalf of **ur**, and the third line pre-computes the rectangle's area and caches it for the benefit of **getArea**.

As it turns out, the above implementation won't even compile. The problem? **ll** and **ur** are full objects embedded within the **rectangle**. **ll** and **ur** aren't pointers to external points—nope, the records making up **ll** and **ur** are wedged inside the **rectangle** like two boxes within a larger one.

C++ constructors are written with the understanding that all direct embedded objects are fully constructed by what's called an initialization list. The initialization list is a comma-delimited list of constructor calls that sits in between the parameter list and the opening curly brace of the constructor's body. Here's the real **rectangle** constructor:

```
rectangle::rectangle(double llx, double lly, double width, double height) :
   ll(llx, lly), ur(llx + width, lly + height)
{
   area = width * height;
}
```

This compiles, because the initialization list specifies exactly how the two embedded **point**s should be constructed. The location of the initialization list suggests that the construction of a **rectangle** is realized as the simultaneous construction of the lower left and upper right corners.

Why is the first version incorrect? I could just say it's because we failed to include **ll** and **ur** on the initialization list, but that's not telling the full story. The more sophisticated explanation: whenever you omit a direct embedded object from the initialization list, the compiler assumes you just want to initialize it using its zero-argument constructor. Since the **point** class doesn't provide a zero-argument constructor (we displaced the compiler-synthesized version when we provided our two-argument one), it's an error to invoke one.

Normally, you only include something on the initialization list if absolutely needs to be there. But you're allowed to list any field—including those that are primitives—on the initialization list if you want to. So while I prefer the first version, we could have written this instead:

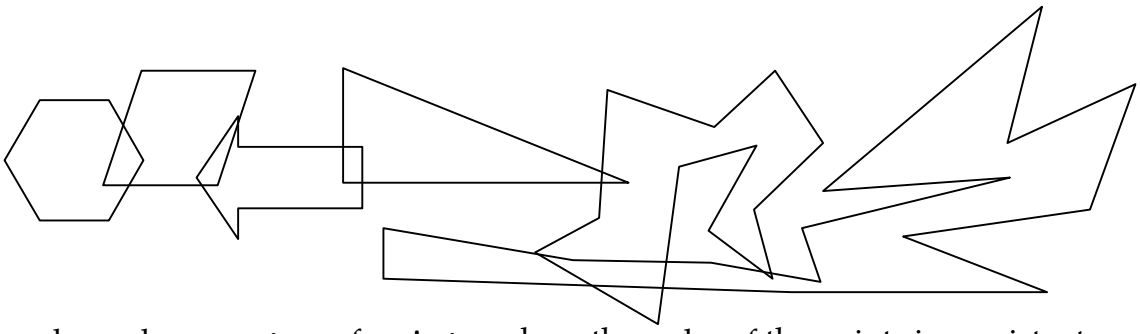
```
rectangle::rectangle(double llx, double lly, double width, double height) :
   ll(llx, lly), ur(llx + width, lly + height), area(width * height) {}
```

Initialization lists are the only avenue for initializing fields that are embedded objects, references, or constants. Otherwise, a field can be initialized within the body of the constructor.

You'll notice that neither the **point** class nor the **rectangle** class provides a destructor. Whenever a class definition omits the destructor from the interface, the compiler synthesizes a **public** destructor with an empty body. In the case of the **point** class, that's just fine: We gain nothing by setting **x** and **y** fields to **0.0** if the memory around those **x** and **y** fields is going away. The compiler-synthesized **rectangle** destructor also does the right thing: it levies the point destructor against the **ur** field, and then levies the same destructor against the **ll** field.

Polygons

The **polygon** class generalizes the notion of a shape in two-dimensional space. In particular, it's designed to model a closed region that's cleanly held by a collection of line segments, as with:



I back the polygon by a **vector** of **point**s, where the order of the points is consistent with the order I'd discover vertices if I walked along its perimeter.

```
class polygon {

   public:
      polygon(const point vertices[], int numVertices);
      polygon(const vector<point>& vertices);

      double getArea() const;
      void translate(double deltax, double deltay);

   private:
      double computeArea() const;
      mutable bool areaPreviouslyComputed;
      mutable double area;
      vector<point> vertices;
};
```

Here's the code for the two constructors:

```
polygon::polygon(const point vertices[], int numVertices) :
   vertices(vertices, vertices + numVertices)
{
   areaPreviouslyComputed = false;
}

polygon::polygon(const vector<point>& vectices) :
   vertices(vertices)
{
   areaPreviouslyComputed = false;
}
```

I used two **vector** constructors (I have no choice: The **vector<point>** is wedged inside the **polygon**) to get most of the work done. Dinkumware (**http://www.dinkumware.com/manuals**) provides some nifty documentation for all of the **vector** constructors. We're only interested in using the sixth and seventh flavors, but here's everything in its full glory:

```
vector();
explicit vector(const Alloc& al);
explicit vector(size_type count);
vector(size_type count, const Ty& val);
vector(size_type count, const Ty& val, const Alloc& al);
vector(const vector& right);
template<class InIt>
    vector(InIt first, InIt last);
template<class InIt>
    vector(InIt first, InIt last, const Alloc& al);
```

> All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument al, if present. For the copy constructor, it is **right.get_allocator()**. Otherwise, it is **Alloc()**.
>
> The first two constructors specify an empty initial controlled sequence. The third constructor specifies a repetition of **count** elements of value **Ty()**. The fourth and fifth constructors specify a repetition of **count** elements of value **val**. The sixth constructor specifies a copy of the sequence controlled by **right**. If **InIt** is an integer type, the last two constructors specify a repetition of **(size_type)first** elements of value **(Ty)last**. Otherwise, the last two constructors specify the sequence **[first, last)**.

Don't worry about **Alloc**, **size_type**, or **explicit**. We'll get there.

The translation process is trivial:

```
void polygon::translate(double deltax, double deltay)
{
   for (unsigned int i = 0; i < vertices.size(); i++) {
      vertices[i].translate(deltax, deltay);
   }
}
```

But computing the area is actually a little bit of work—enough that I've elected to only compute it if **`getArea`** is actually called during runtime. The algorithm for computing a polygon's area is based on Green's Theorem, which itself is the discrete, two-dimensional equivalent of the Stokes' Theorem.

The intention here is to compute the area when and only when we really need it, and to cache the area somewhere so that subsequent calls to **`getArea`** can return the previously computed result. This caching thing—actually quite clever, in my opinion—means that some of the data members with a polygon need to be updated by the implementation of **`getArea`**. Normally, that would means that **`getArea`** would need to be non-**`const`**.

But there's a good reason why **`getArea`** should be **`const`** anyway—it's not necessary for the implementation to change the object, since it could just re-compute the area in a read-only manner every single time. We're **electing** to cache the result, just to make things run faster. And clients of the **`polygon`** class might want to get the area of some constant **`polygon`**, and they'd be blocked if **`getArea`** were left as non-**`const`**.

The **`polygon`** is logically constant—the abstract polygon it represents isn't moving or otherwise changing. But we'd like to be able to change some fields within to cache the area and know we've cached it. In other words, we'd like some fields to be **`mutable`** anyway, even if the object is logically constant.

Enter the **`mutable`** keyword, which can be used to decorate a field within an object. A **`mutable`** field can always be changed, even within the scope of a **`const`** method. You shouldn't use **`mutable`** all that often—only when you need to do something like this:

```
double polygon::getArea() const
{
   if (!areaPreviouslyComputed) {
      area = computeArea();
      areaPreviouslyComputed = true;
   }
   return area;
}

double polygon::computeArea() const
{
   double area = 0.0;
   unsigned int size = vertices.size();
```

```
    for (unsigned int i = 0; i < size; i++) {
        area += 0.5 * (vertices[i].getX() * vertices[(i + 1) % size].getY());
        area -= 0.5 * (vertices[(i + 1) % size].getX() * vertices[i].getY());
    }

    if (area < 0.0) area = -area;
    return area;
}
```