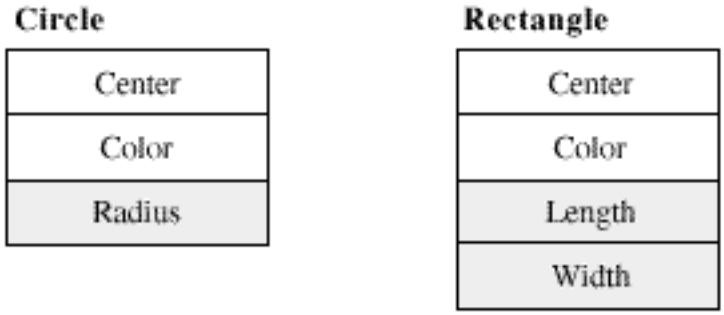
Introduction to Inheritance

There are times when we have a set of related classes for which we would like to share common code. For instance, we might have different shapes which we'll represent as classes. Suppose we have a `Circle` class and a `Rectangle` class. There are certain attributes which are common to circles and rectangles, such as they both might have a point which represents the location of their center, and they both might have a color. However, the dimensions of a circle would probably be defined with a radius, while the dimensions of a rectangle might be defined with a length and width. Conceptually, the objects might look something like this:



Notice how we can arrange the members in the class so that the common elements are in the same location. We could then take the common elements from the `Circle` and `Rectangle` classes and build a common **base class**, which we could call `Shape`. We would then have the `Circle` and `Rectangle` classes extend the `Shape` class. Any code dealing with the common elements of the two shapes would only have to be defined once in the `Shape` class.

Readings from Eckel:

Read Chapter 14, skipping his discussion of operator overloading and multiple inheritance

Readings from Deitel[2]:

Read all of Chapter 9 on Inheritance.

Both texts cover the material very nicely.

The Concept of Subtyping

What we're seeing here is the concept of **subtyping**. In programming languages, A is a subtype of B if and only if A can do everything that B can do. For example, `Circle` would be a subtype of `Shape` because it contains all of the necessary data and function members that the `Shape` class does. This seems a little bit backwards — even though a `Circle` is a **superset** of `Shape`, it is a **subtype** of `Shape` — those wacky programming language theorists!

Programming languages which support subtyping will allow an object of a subtype to be assigned to an object of its supertype without casting or conversion. You can also pass a subtype as a parameter to a function which is expecting its supertype. In concrete terms, this means that you can assign a `Circle` to a `Shape`, or pass a `Rectangle` to a function expecting a `Shape`. C++ supports this, but C would force you to do explicit type casting.

C++ Inheritance

Subtyping in C++ is provided through **inheritance**. The inheritance mechanism in C++ is provided as part of the class definition, and it dictates what is to be shared between a set of classes. In general, both the interface and the implementation of a **base class** are inherited by a **derived class**. A base class defines a base set of functions and data, and a derived class will generally extend the base class with its own functions and/or data. In C++ base classes are also referred to as superclasses, and derived classes are also referred to as subclasses. C++ permits multiple levels of inheritance. For example, we could have a class `Microbus` which inherits from a class `VW` which, in turn, inherits from a class `Automobile`. Inheritance is transitive—the `Microbus` class will be a subtype of `VW`, as well as a subtype of `Automobile`. Because a base class can have multiple derived classes, it is common to call a base class and its set of derived classes a **class tree** or a **class hierarchy**.

Inheritance Syntax

Suppose we were implementing our `Shape`, `Circle`, and `Rectangle` classes. First, we must define our base class:

```
class Shape
{
   public:
      Shape(const Point& location, const Color& color);
      ~Shape();
      ...

   private:
      Point location;
      Color color;
};
```

This provides the basic set of operations which is common to all shapes we'll be defining. Now, in order to extend the `Shape` class to make a `Circle`, we would do this:

```
class Circle : public Shape
{
   public:
      Circle(const Point& location, const Color& color, double radius);
      ~Circle();
      ...

   private:
      double radius;
};
```

The "`: public Shape`" in the class definition states that the `Circle` will inherit all of the functions and data from the `Shape` class. However, the `Circle` class can only access the public members of the `Shape` class. The `Shape` class knows nothing about the `Circle` class, and only has access to its own members.

The `Circle` class can re-define any methods in the `Shape` class, but that will shadow any inherited methods with the same name. In other words, function overloading only applies within one level of a class hierarchy.

Upcasting

As mentioned before, a `Circle` object can be implicitly converted into a `Shape` object, because `Circle` is a subtype of `Shape`. This is called upcasting in C++, because you're moving up in the class hierarchy. The following code is legal, but not necessarily a good idea:

```
Circle circle(pt, "Red", 5);
Shape shape = circle;
```

What happens here is that shape loses all information about circle which isn't contained within the `Shape` class. Namely, it loses the `radius` data member. This is known as **object slicing**. Even if we casted `shape` back to a `Circle` object, we wouldn't get back the radius that we lost. To avoid object slicing, you should only (with few exceptions) use upcasting with pointers or references. This includes using pointers or references when passing objects to functions and methods, and when returning objects.

Constructors and Destructors

Constructors and destructors have interesting behavior when combined with inheritance. Generally, the derived class will call the constructor of its base class as part of its constructor.

This is done using as part of the constructor initialization list. For example, the `Circle` constructor would look like this:

```
Circle::Circle(const Point& loc, const Color& color,
                                 double radius) : Shape(loc, color)
{
   this->radius = radius;
}
```

Instead of initializing a single member in the initialization list, you call the parent class constructor with all necessary parameters. You can mix the initialization of members with the initialization of the base class. By supplying the parent class constructor in the initialization list, you guarantee that the base class constructor is called **before** the derived class constructor. Destructors are called in the reverse order of constructors, and they are called automatically. For example, when destroying a `Circle` object, first the `Circle` destructor will be called, then the `Shape` destructor.

Assignment Operator

The assignment operator is not inherited by a derived class, even if the base class provides a non-default assignment operator. This is because the base class assignment operator knows nothing about the derived class. For instance, if the `Shape` class had an assignment operator, it wouldn't know anything about the radius of a `Circle`, or the length and width of a `Rectangle`, so it wouldn't know to copy them. Thus, if you must provide an assignment operator for the base class, you'll have to provide an assignment operator for every derived class. You can call the superclass version, but you can't rely on it to do everything.

Overriding Methods

As mentioned earlier, a derived class does not have to strictly add functions in order to extend its base class. It can also redefine, or **override** a method from its base class. For example, suppose that we wanted to be able to determine if a given point was inside of a `Shape`. For the general `Shape` class, we might define a `containsPoint` method which tests to see if the point is inside a box which entirely contains the shape[1]. For the `Circle` class, we can use our friend the Pythagorean theorem to determine this, and for the `Rectangle` we can compare it with the width and height.

Calling Overridden Methods

Often, it is useful to share code by implementing a method within a derived class by using the code from the same method within the base class. For example, suppose we had a `ComplexPolygon` object which could represent shapes that looked like these:

[1] It may not make sense why we'd even provide this routine for the `Shape` class, but it will make more sense after our next lecture.

As you might imagine, it could take a long time to figure out if a point was located in one of these shapes. However, the `Shape` base class will tell us if the given point was inside the shape's bounding box. This is a crude, but very fast approximation. We know that if a point lies outside of the shape's bounding box, there is no possible way that it could lie inside of the shape. We can use the `Shape` class `containsPoint` method by using the `Shape` class specifier:

```
bool ComplexPolygon::containsPoint(Point& pt)
{
   if(!Shape::containsPoint(pt)) // Call the base class method
      return false;

   Do the precise check to see if pt is within the polygon
}
```

You can access any non-`private` method within a base class by using the class specifier. It looks somewhat like calling a `static` class method, but it can be used even for calling non-`static` class methods.

`protected` Members

So far, we've only seen two access specifiers within classes: `public` and `private`. Now that we've seen inheritance, it makes sense to introduce the third type of access specifier. The `protected` class specifier gives `public` access to all methods within derived classes, but `private` access to all other functions. It is still best to keep all data `private`—don't be tempted to make it `protected`. However, there are some functions which can be declared as `protected`:

- accessors and mutators that aren't meant to be `public`, but that you want derived classes to be able to use
- other useful helper functions which can be used by derived classes.

Protected and Private Inheritance

In C++ there are three types of inheritance—`public`, `protected`, and `private`. To use `protected` or `private` inheritance, you would substitute `public` with the desired form of inheritance when declaring the derived class. In English, this means that instead of deriving like this:

```
class Circle : public Shape
{
   ...
};
```

you would derive like this:

```
class Circle : private Shape
{
   ...
};
```

The difference between these forms of inheritance specifies who can treat `Circle` as a subtype of `Shape`:

- For `public` inheritance, any function can treat `Circle` as a subtype of `Shape`.
- For `protected` inheritance, any method within the `Circle` class or a class derived from `Circle` can treat `Circle` as a subtype of `Shape`.
- For `private` inheritance, only methods within the `Circle` class can treat `Circle` as a subtype of `Shape`.

These other forms of inheritance are used when you want to reuse code, but there is no truly logical relationship between the two types. `public` inheritance is by far the most common type.