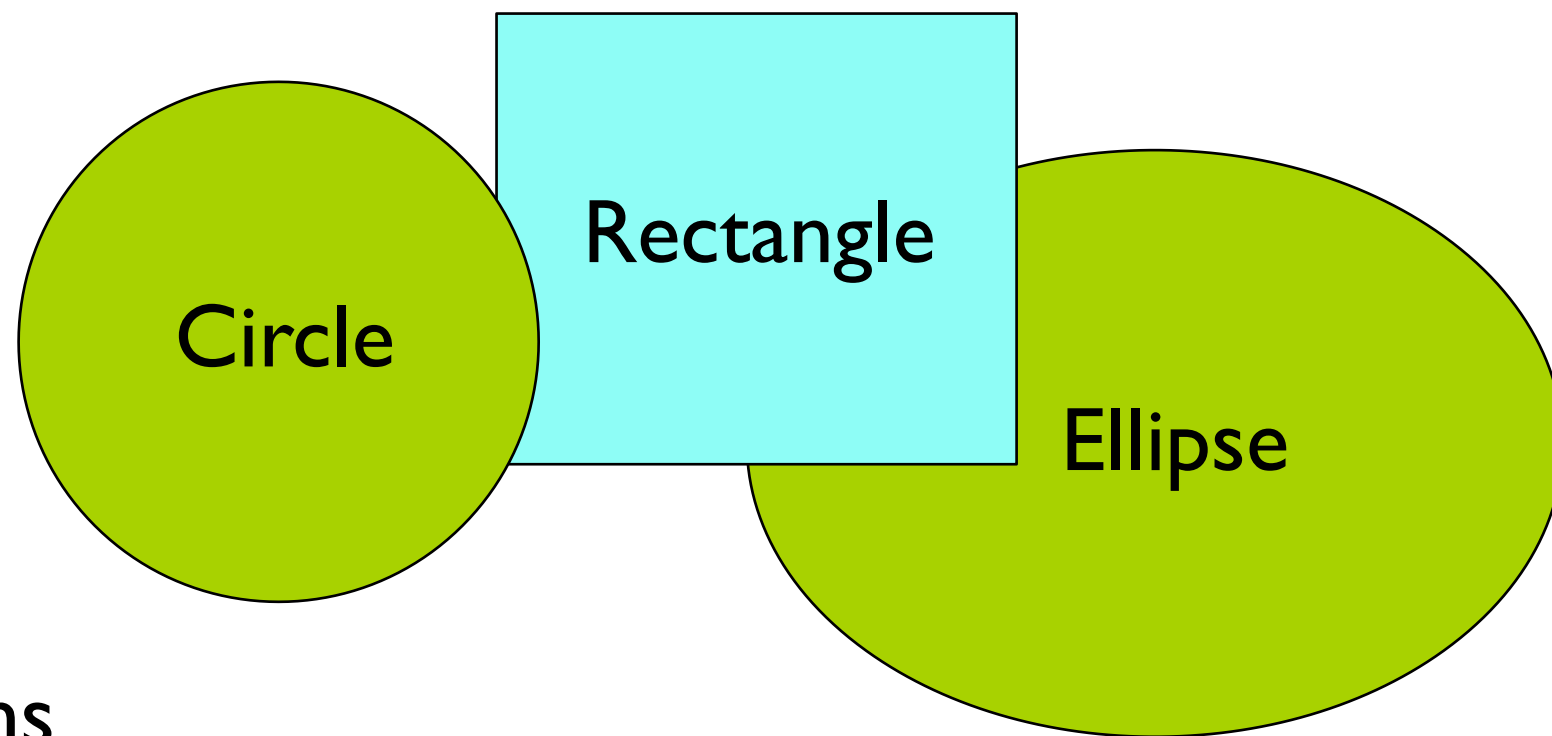


Polymorphism

Object-Oriented Programming with C++

A drawing program



Operations

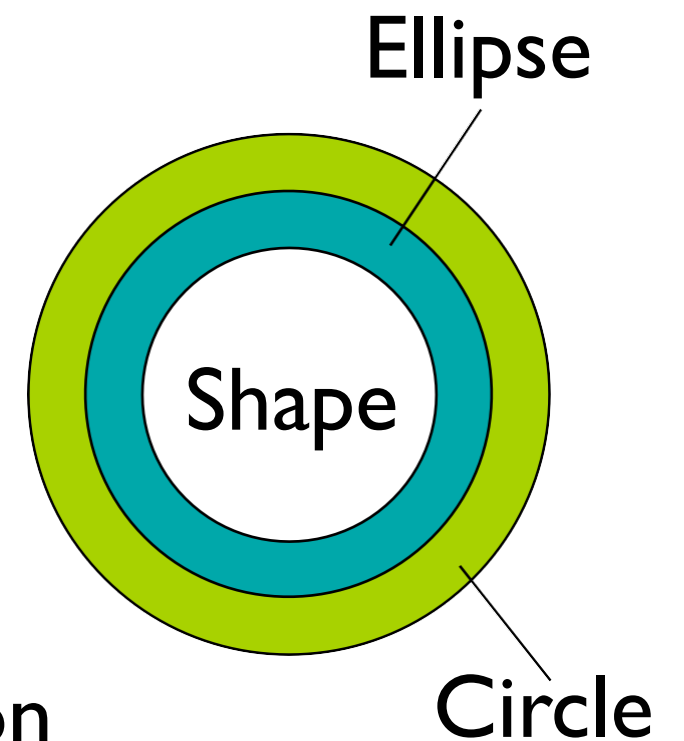
- render
- move
- resize

Data

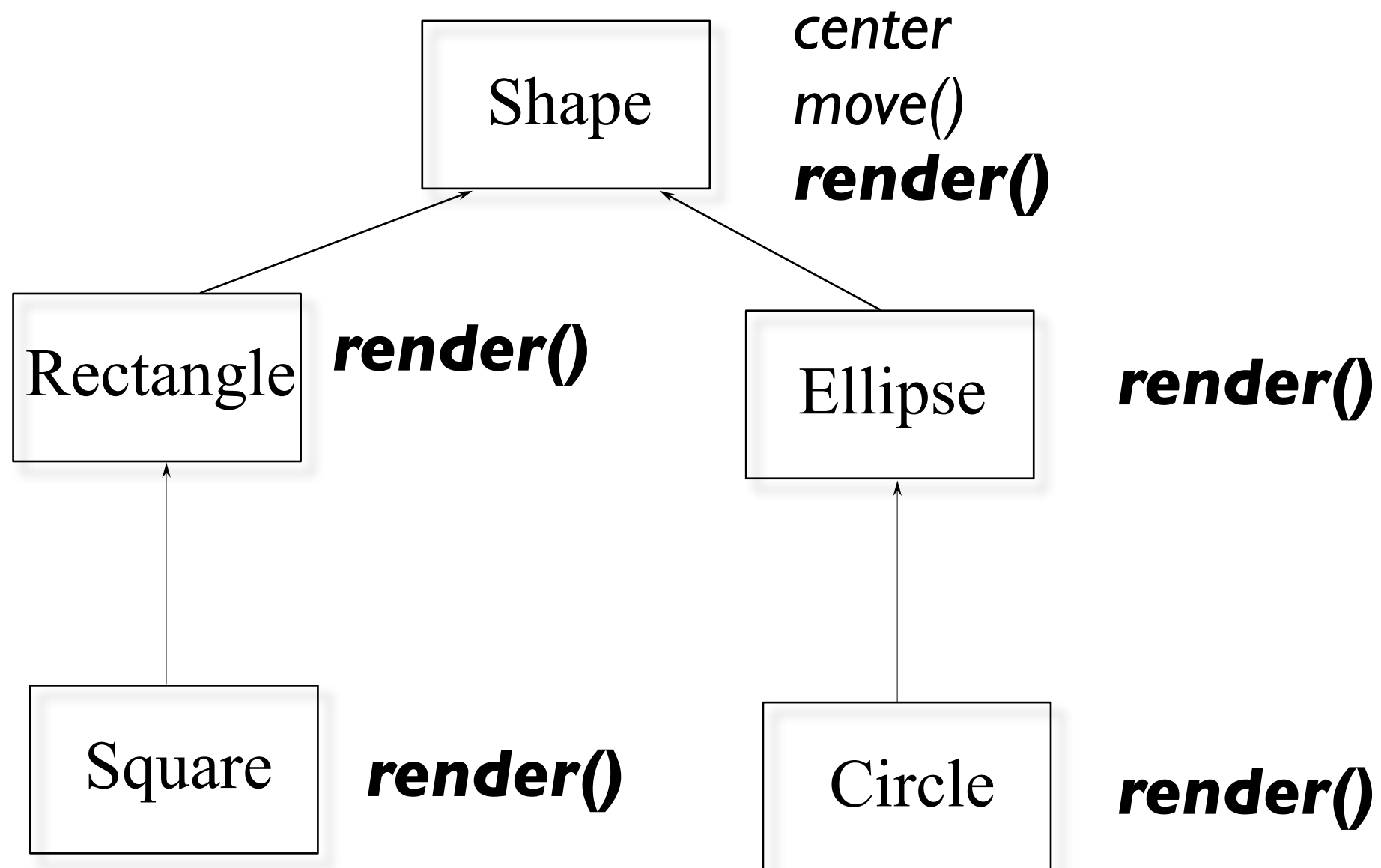
+ center

Inheritance in C++

- Can define one class in terms of another
- Can capture the notion that
 - An ellipse is a shape
 - A circle is a special kind of ellipse
 - A rectangle is a different shape
 - Circles, ellipses, and rectangles share common
 - attributes
 - services
 - Circles, ellipses, and rectangles are not identical



Conceptual model



Note: Deriving Circle from Ellipse is a poor design choice!

Shape

- Define the general properties of a Shape

```
class Point {...};           // (x,y) point
class Shape {
public:
    Shape();
    void move(const Point&);
    virtual void render();
    virtual void resize();
    virtual ~Shape();
protected:
    Point center;
}
```

Add new shapes

```
class Ellipse: public Shape {
public:
    Ellipse(float major, float minor);
    virtual void render(); // will define own
protected:
    float major_axis, minor_axis;
};

class Circle: public Ellipse {
public:
    Circle(float radius) : Ellipse(radius, radius) {}
    virtual void render();
};
```

Example

```
void render(Shape* p) {  
    p->render();          // calls correct render function  
}                          // for given Shape!
```

```
void func() {  
    Ellipse ell(10, 20);  
    ell.render();  
    Circle circ(40);  
    circ.render();  
    render(&ell);  
    render(&circ);  
}
```

Example

```
void render(Shape* p) {  
    p->render();          // calls correct render function  
}                          // for given Shape!
```

```
void func() {  
    Ellipse ell(10, 20);  
    ell.render();  
    Circle circ(40);  
    circ.render();  
    render(&ell);  
    render(&circ);  
}
```

code & demo

Polymorphism

Polymorphism

- Upcast: take an object of the derived class as an object of the base one.
 - Ellipse can be treated as a Shape

Polymorphism

- Upcast: take an object of the derived class as an object of the base one.
 - Ellipse can be treated as a Shape
- Dynamic binding:
 - Binding: which function to be called
 - Static binding: call the function as the declared type
 - Dynamic binding: call the function according to the “real” type of the object

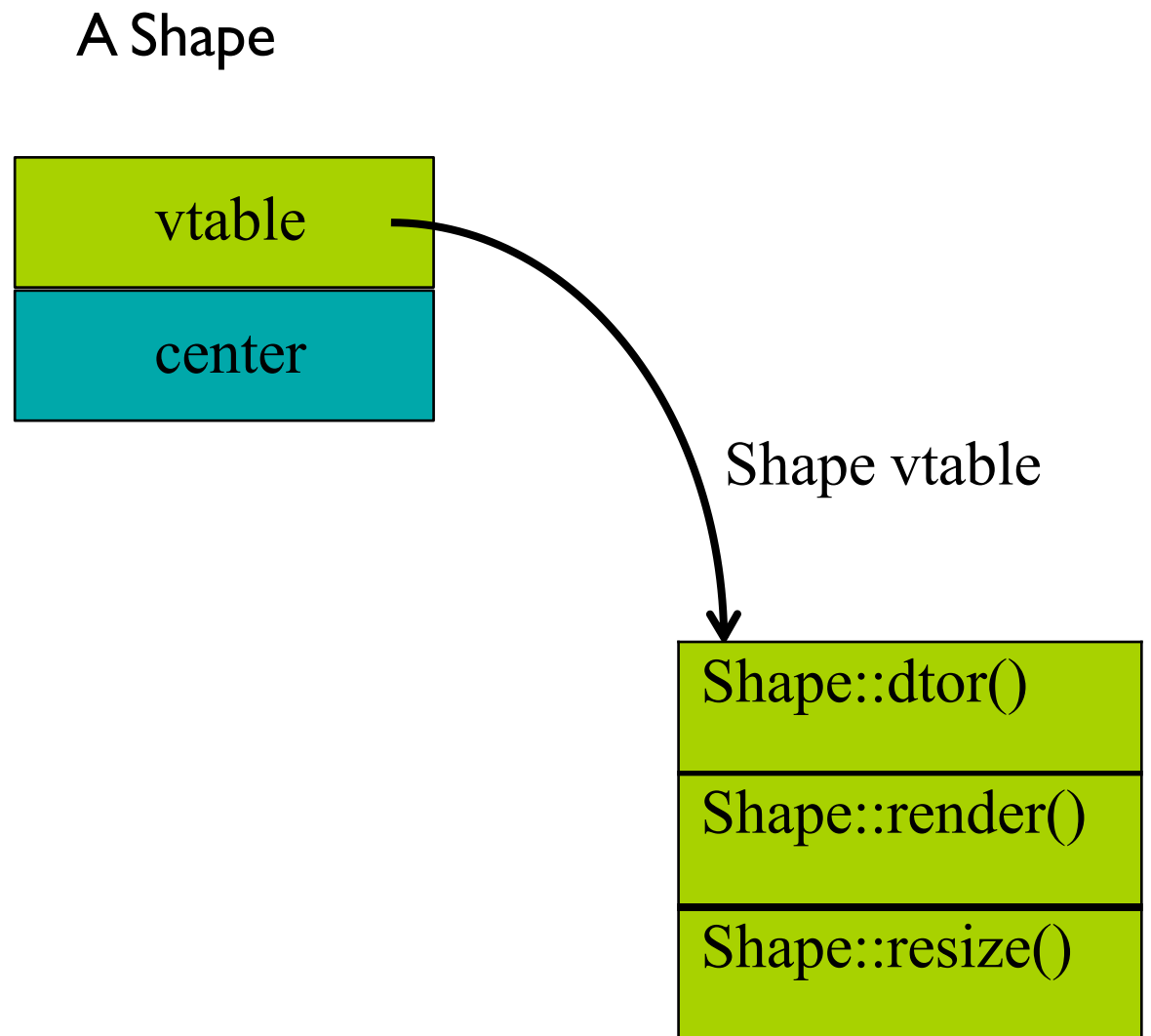
How virtual works in C++

```
class Shape {  
public:  
    Shape();  
    virtual ~Shape();  
    virtual void render();  
    void move(const  
        Point&);  
    virtual void resize();  
protected:  
    Point center;  
};
```

code & demo

How virtual works in C++

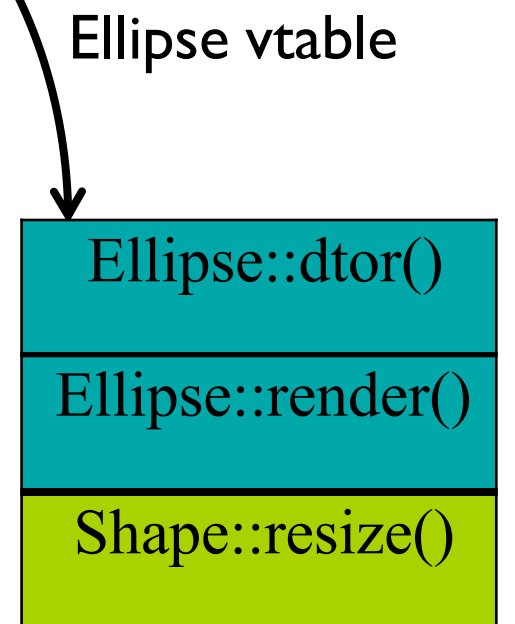
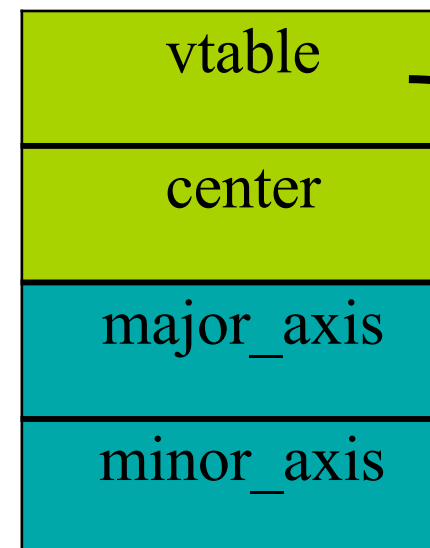
```
class Shape {  
public:  
    Shape();  
    virtual ~Shape();  
    virtual void render();  
    void move(const  
        Point&);  
    virtual void resize();  
protected:  
    Point center;  
};
```



Ellipse

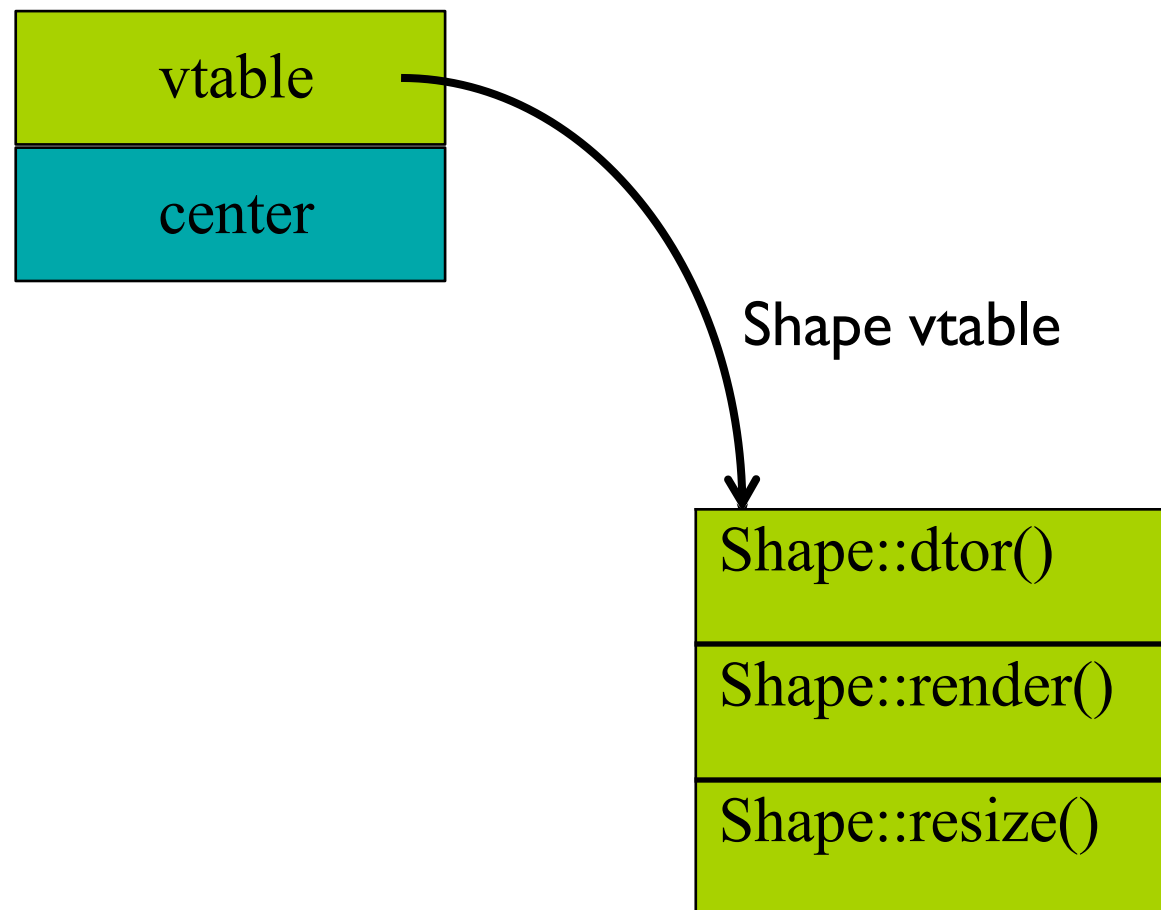
```
class Ellipse: public Shape{  
public:  
    Ellipse(float major,  
            float minor);  
    virtual void render();  
  
protected:  
    float major_axis,;  
    float minor_axis;  
};
```

An Ellipse

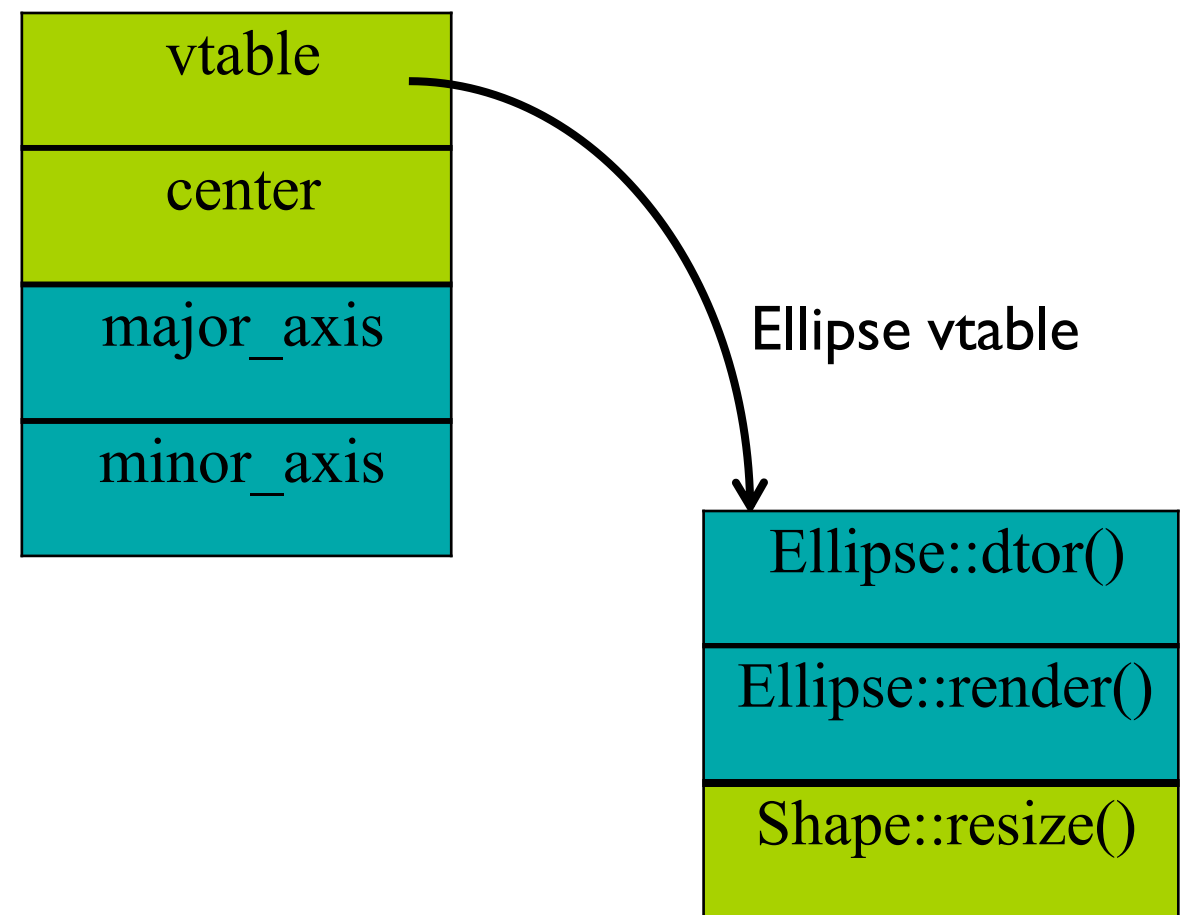


Shape vs. Ellipse

A Shape



An Ellipse

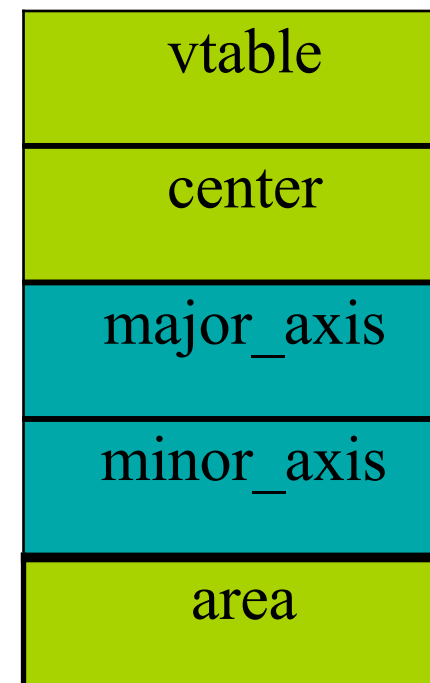


Circle

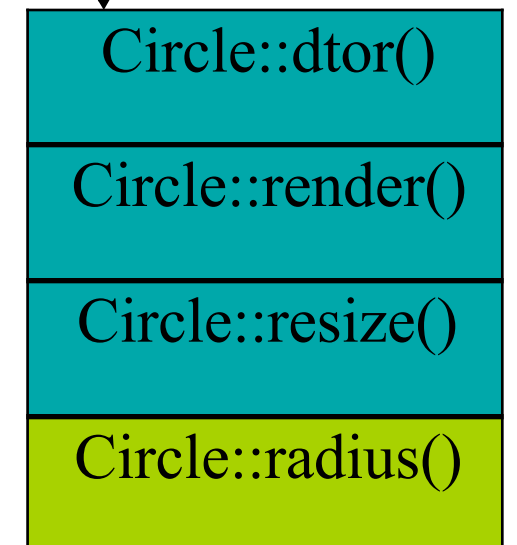
```
class Circle: public Ellipse{  
  
public:  
    Circle(float radius);  
    virtual void render();  
    virtual void resize();  
    virtual float radius();  
  
protected:  
    float area;  
  
};
```

code & demo

A Circle



Circle vtable



What happens if

What happens if

```
Ellipse elly(20F, 40F);
```

```
Circle circ(60F);
```

```
elly = circ; // ?
```

code & demo

What happens if

```
Ellipse elly(20F, 40F);
```

```
Circle circ(60F);
```

```
elly = circ;
```

- **Area of `circ` is sliced off**
 - (Only the part of `circ` that fits in `elly` gets copied)

What happens if

```
Ellipse elly(20F, 40F);
```

```
Circle circ(60F);
```

```
elly = circ;
```

- Area of `circ` is sliced off
 - (Only the part of `circ` that fits in `elly` gets copied)
- Vptr from `circ` is ignored; the vptr in `elly` points to the Ellipse vtable

```
(&elly)->render(); // Ellipse::render()
```

What happens with pointers?

What happens with pointers?

```
Ellipse* elly = new Ellipse(20F, 40F);
```

```
Circle* circ = new Circle(60F);
```

```
elly = circ;
```

What happens with pointers?

```
Ellipse* elly = new Ellipse(20F, 40F);  
Circle*  circ = new Circle(60F);  
elly = circ;
```

- Well, the original Ellipse for `elly` is lost....

What happens with pointers?

```
Ellipse* elly = new Ellipse(20F, 40F);  
Circle* circ = new Circle(60F);  
elly = circ;
```

- Well, the original Ellipse for `elly` is lost....
- `elly` and `circ` point to the same Circle object!

```
elly->render(); // Circle::render()
```


Virtual and reference arguments

Virtual and reference arguments

```
void func(Ellipse& elly) {  
    elly.render();  
}
```

```
Circle circ(60F);  
func(circ);
```

Virtual and reference arguments

```
void func(Ellipse& elly) {  
    elly.render();  
}
```

```
Circle circ(60F);  
func(circ);
```

- References act like pointers

Virtual and reference arguments

```
void func(Ellipse& elly) {  
    elly.render();  
}
```

```
Circle circ(60F);  
func(circ);
```

- References act like pointers
- Circle::render() is called

Virtual destructors

- Make destructors ***virtual*** if they might be inherited

```
Shape *p = new Ellipse(100.0F, 200.0F);
```

```
...
```

```
delete p;
```

- If `Shape::~Shape()` is not virtual, only `Shape::~Shape()` will be invoked!
- Want `Ellipse::~~Ellipse()` to be called
 - Must declare `Shape::~Shape()` virtual
 - It will call `Shape::~Shape()` internally

Overriding

- Overriding redefines the body of a virtual function

```
class Base {  
public:  
    virtual void func();  
}  
class Derived : public Base {  
public:  
    void func() override;  
    //overrides Base::func()  
}
```

Calls up the chain

- You can still call the overridden function for *reuse*:

```
void Derived::func() {  
  
    cout << "In Derived::func!";  
  
    Base::func(); // call to base class  
}
```

- This is a common way to add new functionality
- No need to copy the old stuff!

Return types relaxation (current)

- Suppose D is publicly derived from B
- $D :: f()$ can return a subclass of the return type defined in $B :: f()$
- Applies to pointer and reference types
 - e.g. $D\&$, D^*
- In most compilers now

Relaxation example

```
class Expr{
public:
    virtual Expr* newExpr();
    virtual Expr& clone();
    virtual Expr self();
};

class BinaryExpr: public Expr{
public:
    virtual BinaryExpr* newExpr(); // ok
    virtual BinaryExpr& clone();    // ok
    virtual BinaryExpr self();      // Error!
};
```

Overloading and virtual

- Overloading adds multiple signatures

```
class Base {  
    public:  
        virtual void func();  
        virtual void func(int);  
};
```

- If you *override* an *overloaded* function, you must override all of the variants!
 - Can't override just one
 - If you don't override all, some will be hidden

Overloading example

- When you *override* an *overloaded* function, override all of the variants!

```
class Derived: public Base{
public:
    virtual void func() {
        Base::func();
    }
    virtual void func(int) { ... };
}
```

Abstract classes

- Why use them?
 - Modeling
 - Force correct behavior
 - Define interface without defining an implementation
- When to use them?
 - Not enough information is available
 - When designing for interface inheritance

Protocol / Interface classes

- Abstract base class with
 - All non-static member functions are *pure virtual* except destructor
 - Virtual destructor with empty body
 - No non-static member variables, inherited or otherwise
 - May contain static members

Example interface

- Unix character device

```
class CDevice {  
public:  
    virtual ~CDevice() {}  
  
    virtual int read(...) = 0;  
    virtual int write(...) = 0;  
    virtual int open(...) = 0;  
    virtual int close(...) = 0;  
    virtual int ioctl(...) = 0;  
};
```