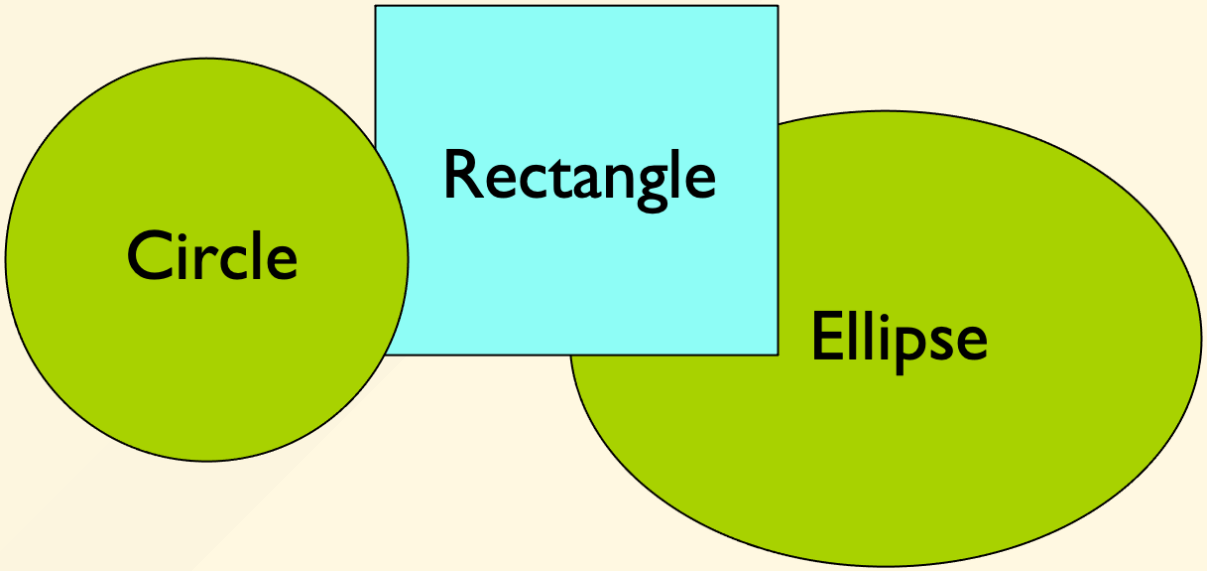
Polymorphism

Object-Oriented Programming with C++

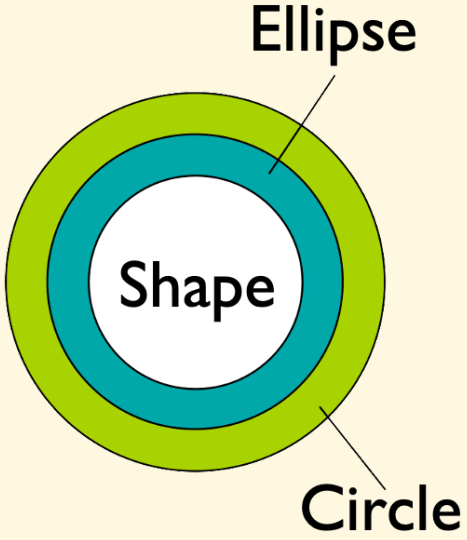
A drawing program

- Operations + Data
  - render, move, resize, ...
  - center, ...

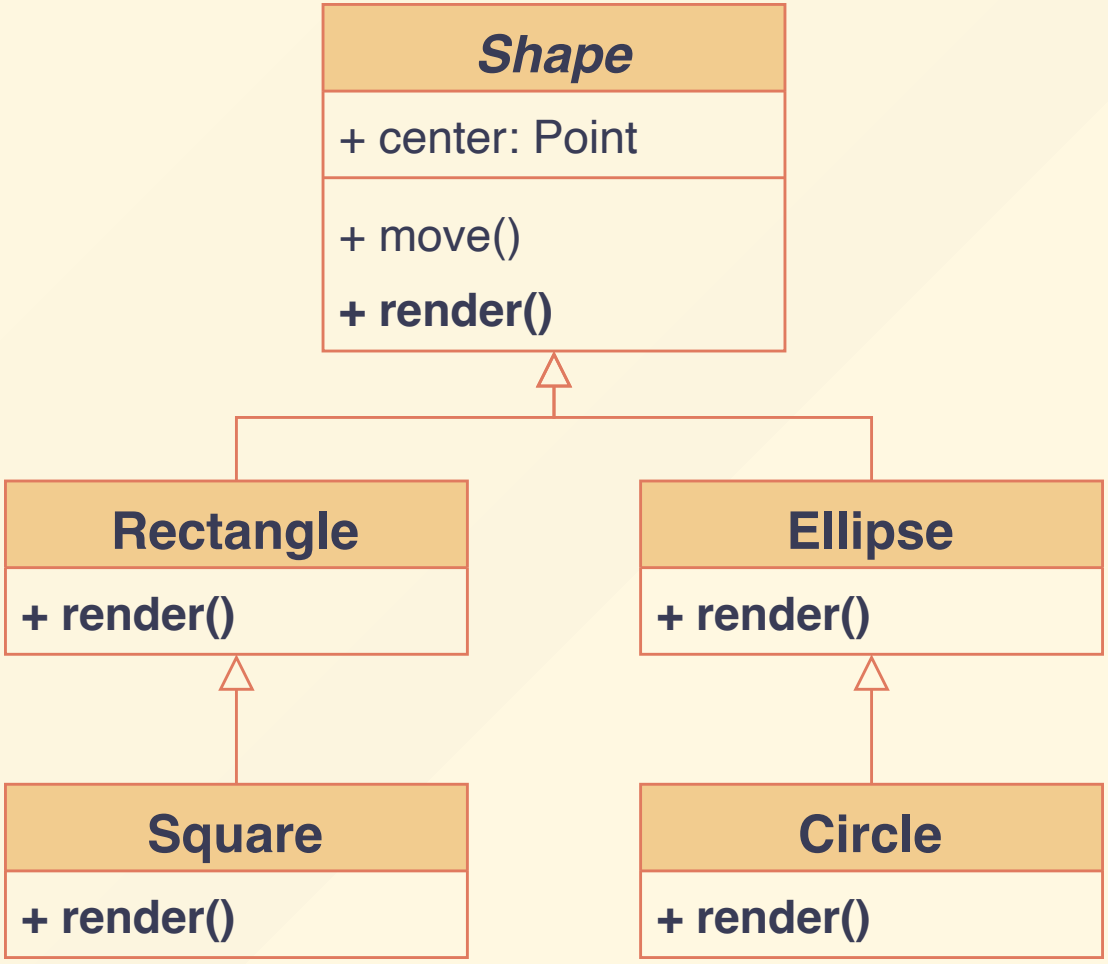


Inheritance in C++

- Capture the notion that
  - An *ellipse* is a shape
  - A *circle* is a special ellipse
  - A *rectangle* is a different shape
  - They share common
    - attributes
    - services



Conceptual model



Shape

Define the general properties of a Shape

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

Add new shapes

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

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

Usage

```
void render(Shape* p){
  p->render(); // calls given-shape's render()
}

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

Polymorphism

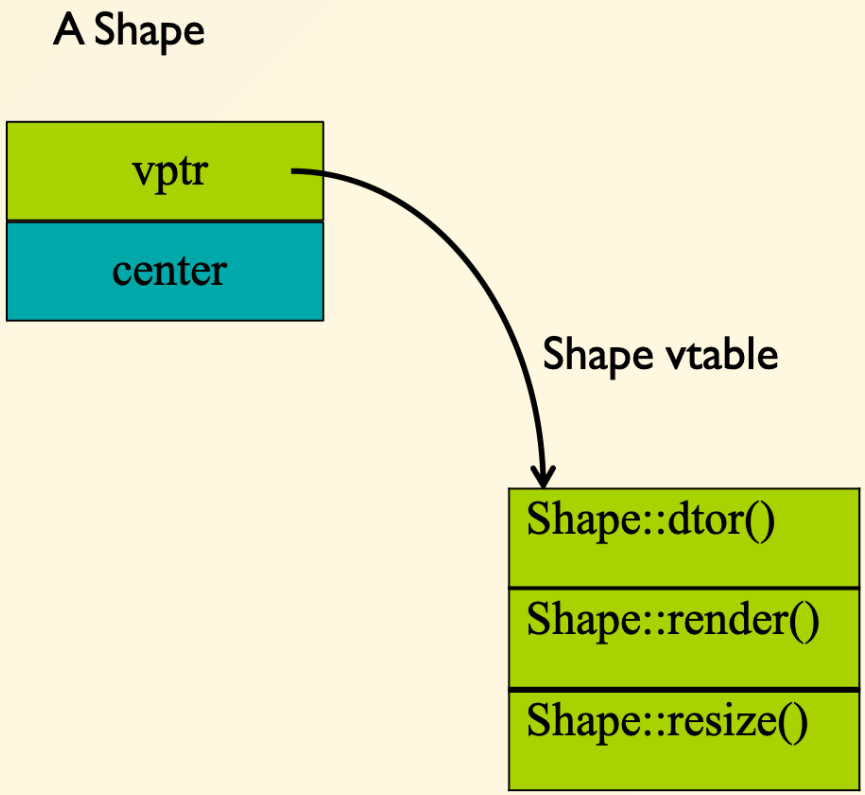
- Upcast: take an object of the derived class as an object of the base one.
  - Ellipse can be treated as a Shape
- Binding: which function to be called
  - Static binding:
    - call the function as the declared type
  - Dynamic binding:
    - call the function according to the *real* type of the object

How `virtual` works in C++

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

How `virtual` works in C++

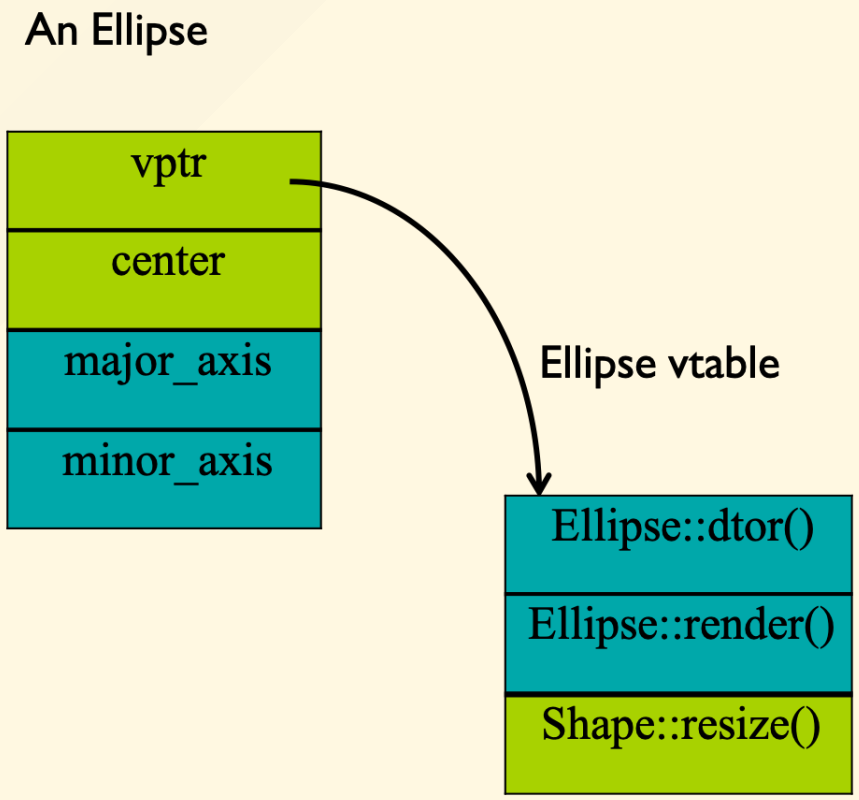
```
class Point {...};
class Shape {
public:
  Shape();
  virtual ~Shape();
  virtual void render();
  virtual void resize();
  void move(
    const Point&);
protected:
  Point center;
};
```



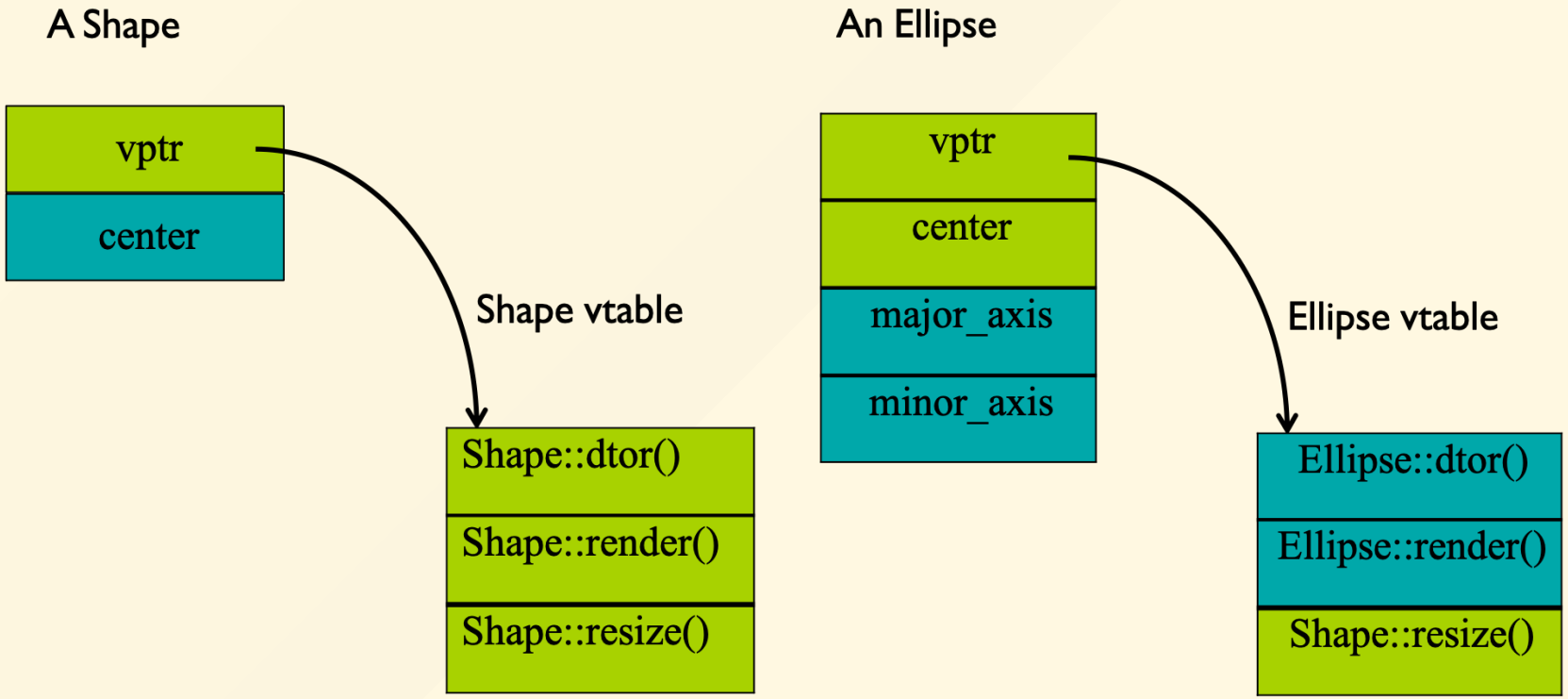
Ellipse

```
class Ellipse
  : public Shape
{
public:
   Ellipse(float major,
           float minor);
   ~Ellipse();
   virtual void render();

protected:
   float major_axis;
   float minor_axis;
};
```



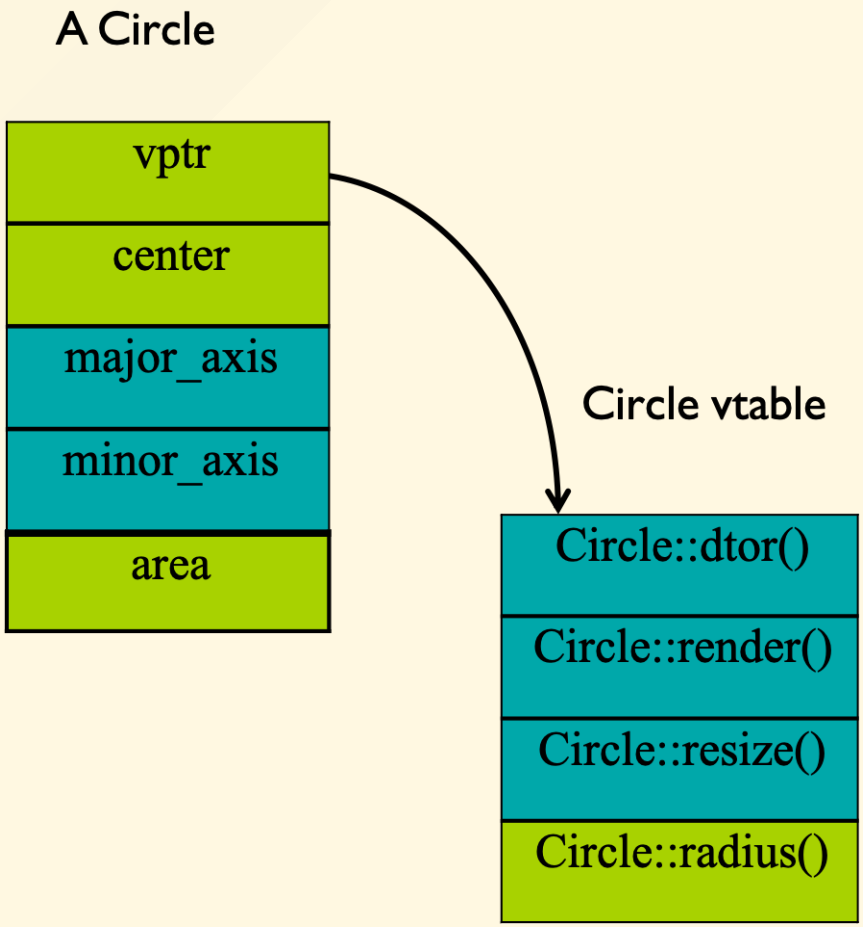
Shape vs. Ellipse



Circle

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

protected:
   float area;
};
```



What happens if

```
Ellipse elly(20f, 40f);
Circle circ(60f);
elly = circ;  // ???
```

What happens if

```
Ellipse elly(20f, 40f);
Circle circ(60f);
elly = circ;
```

- Area of `circ` is sliced off
  - only the part of `circ` fits in `elly` gets copied
- The *vptr* from `circ` is ignored
  - *vptr* in `elly` still points to the Ellipse vtable

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

```
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;  // which dtor?
```

- `Shape::~Shape()` is invoked if not virtual!
- Want `Ellipse::~Ellipse()` to be called:
  - Must declare `virtual Shape::~Shape()`, which is implicitly called inside `Ellipse::~Ellipse()`.

Overriding

`override` 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

- 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*`

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: 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!
  - If not, others will be hidden

Abstract classes

- Why shall we use them?
  - Modeling, force correct behavior
  - Define interface without defining an implementation
- When to use them?
  - Not enough information is available
  - 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
  - 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;
};
```