Object－Oriented Programming

Week 8

Polymorphism

Weng Kai

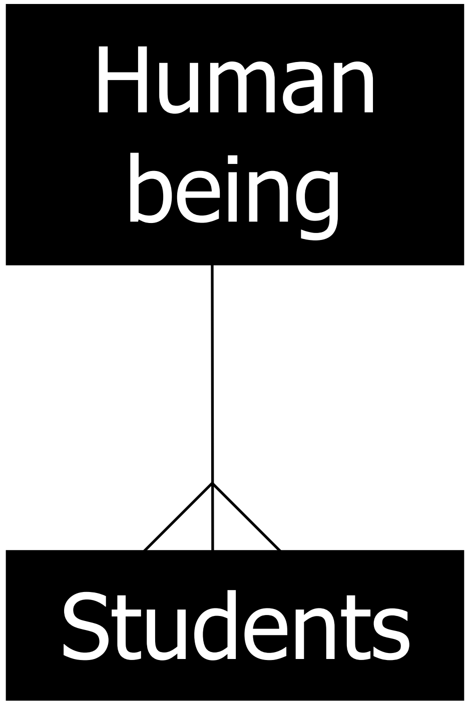
Conversions 转换

- Public Inheritance should imply substitution
  - If B *isa* A, you can use a B anywhere an A can be used.
    - if B isa A, then everything that is true for A is also true of B.
  - Be careful if the substitution is not valid!

D is derived from B		
D	⇒	B
D*	⇒	B*
D&	⇒	B&

Up-casting 向上造型

- Is to regard an object of the derived class as an object of the base class.
- It is to say: Students are human beings. You are students. So you are human being.

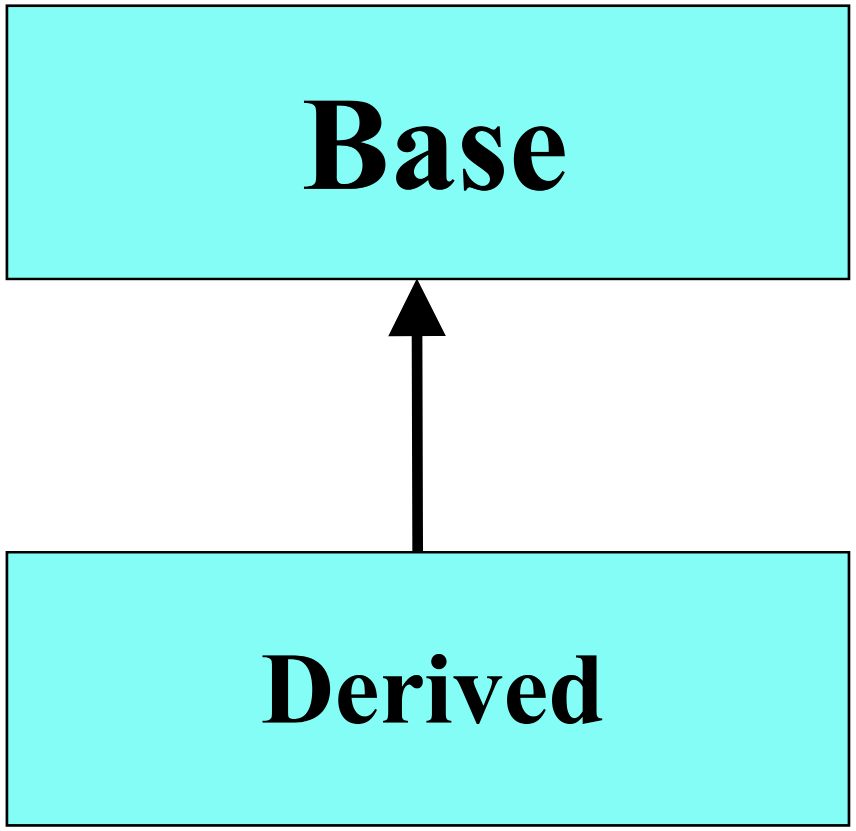


Human being

Students

Upcasting

- Upcasting is the act of converting from a Derived reference or pointer to a base class reference or pointer.



Upcasting examples

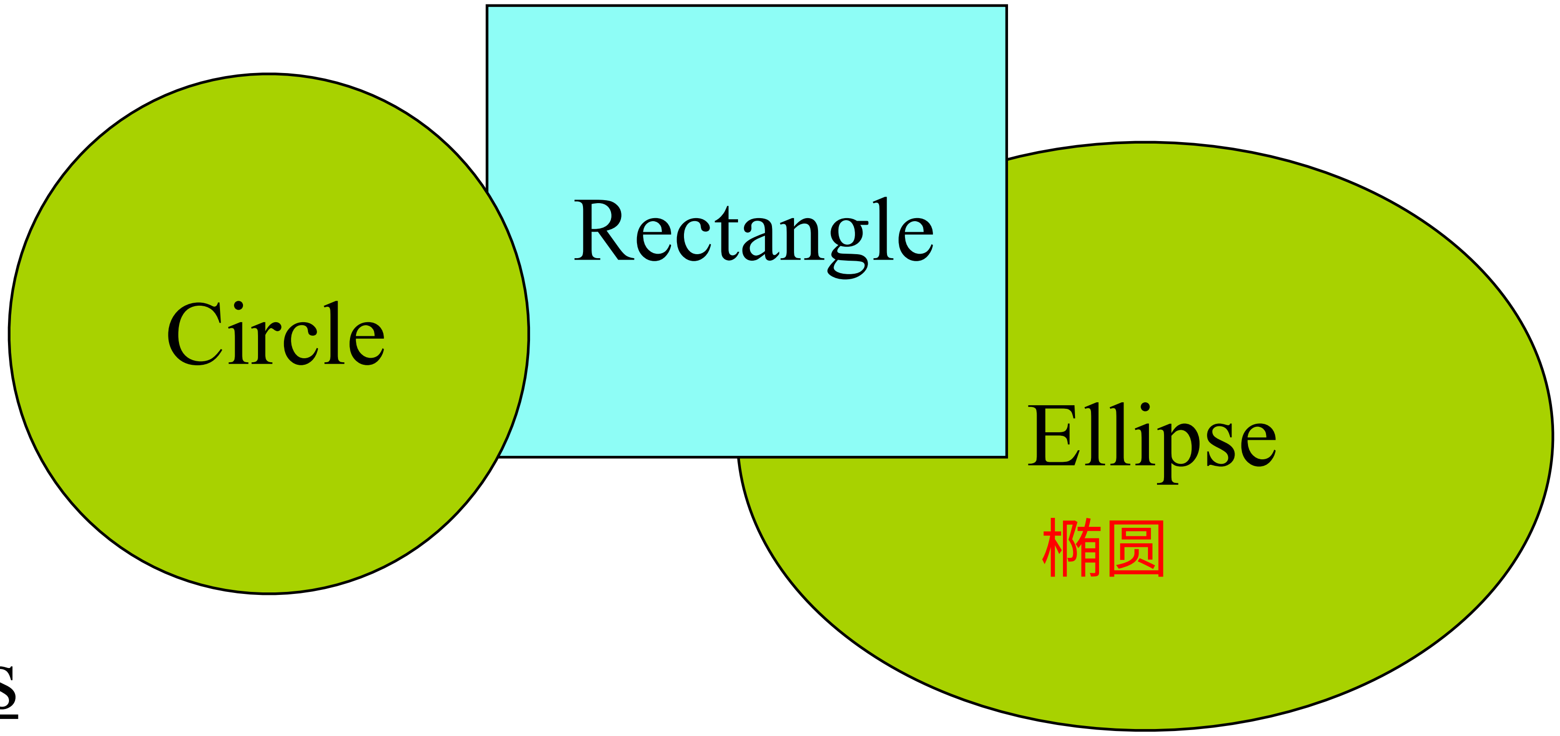
```
Manager pete( "Pete", "444-55-6666", "Bakery");
Employee* ep = &pete;  // Upcast
Employee& er = pete;    // Upcast
```

- Lose type information about the object:

```
  ep->print( cout );    //  prints base class version
```

前提是父类中的print前没有virtual，当有virtual时，
还是会调用对应子类的方法

A drawing program



Operations

- render
- move
- resize

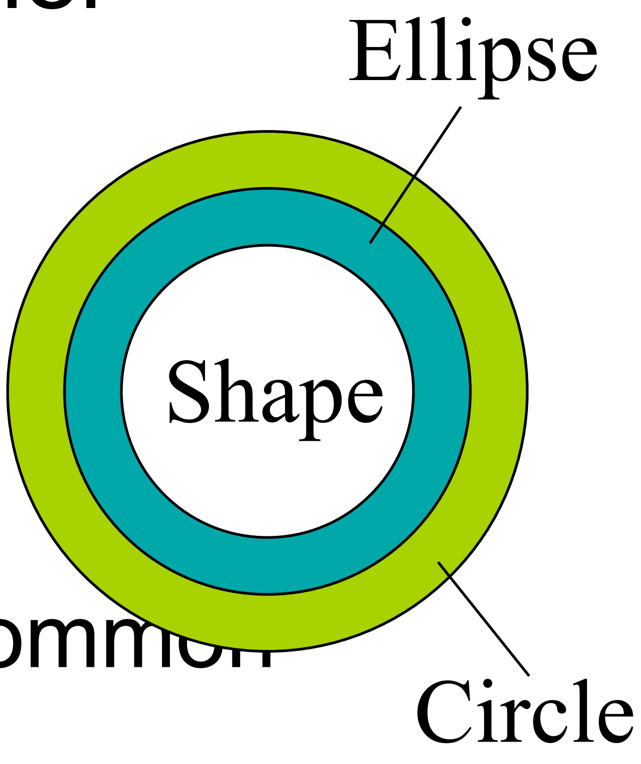
根据数学表达式绘制图形

Data

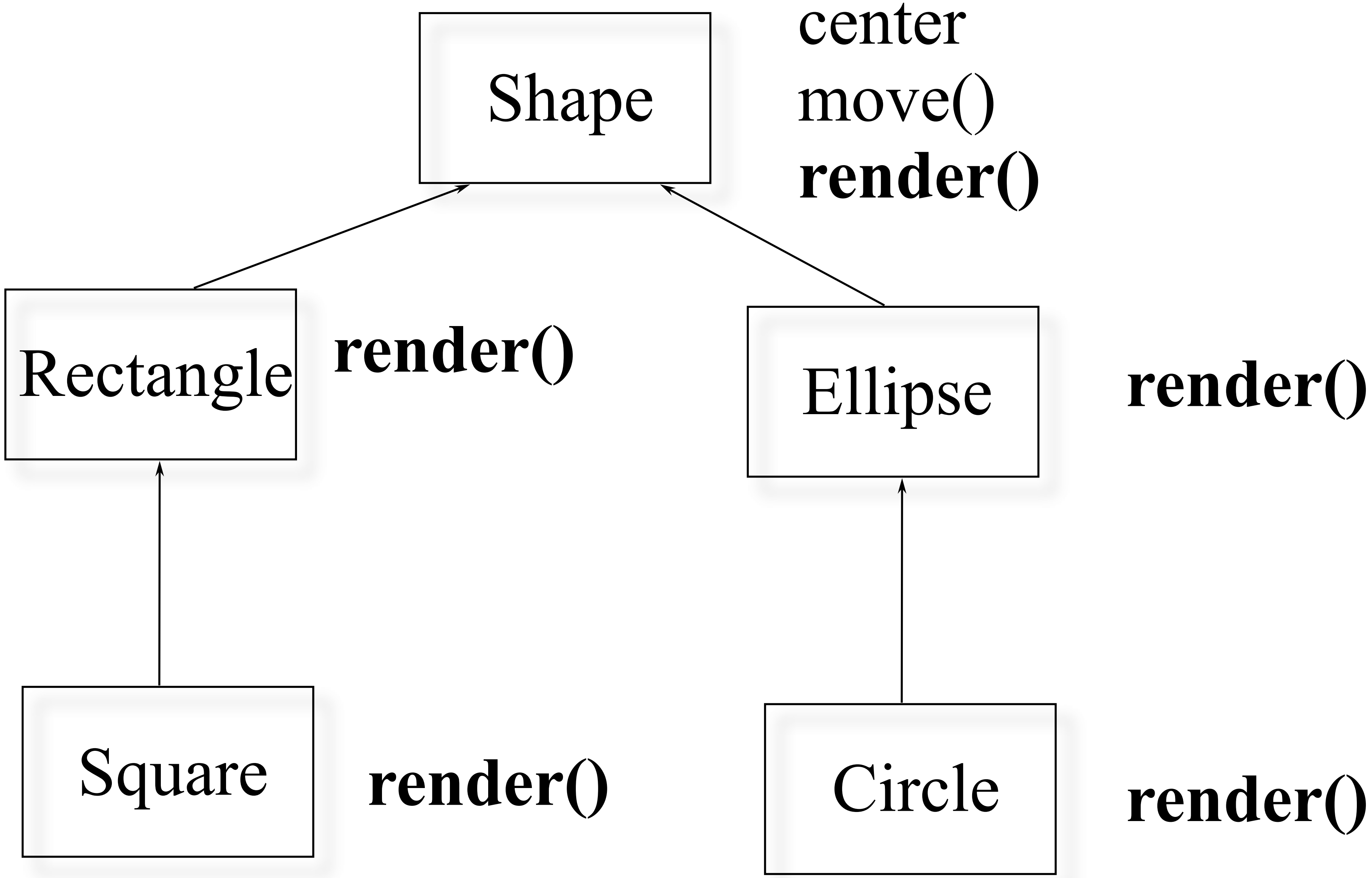
+ center

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



Shape

center
move()
render()

Rectangle

render()

Ellipse

render()

Square

render()

Circle

render()

椭圆参数：a,b
圆形：r
圆形参数少

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

In C++

- Define the general properties of a Shape

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

无virtual，子类中没有重写

Add new shapes

```
class Ellipse : public Shape {
public:
  Ellipse(float maj, float minr);
  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();
};
```

有自己的render，也声明virtual，说明子类也可能重写该方法

p是多态变量，静态类型时shape，动态类型时对应运行时的子类

向上造型

Example

```
void render(Shape* p) {
  p->render();    // calls correct render function
}                 // for given Shape!
void func() {
  Ellipse ell(10, 20);
  ell.render();    // static -- Ellipse::render();

  Circle circ(40);
  circ.render();  // static -- Circle::render();

  render(&ell);    // dynamic -- Ellipse::render();
  render(&circ);  // dynamic -- Circle::render()
}
```

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 code
    - Dynamic binding: call the function of the object

指针+virtual或者引用+virtual才会动态绑定，其他都是静态绑定

Virtual functions

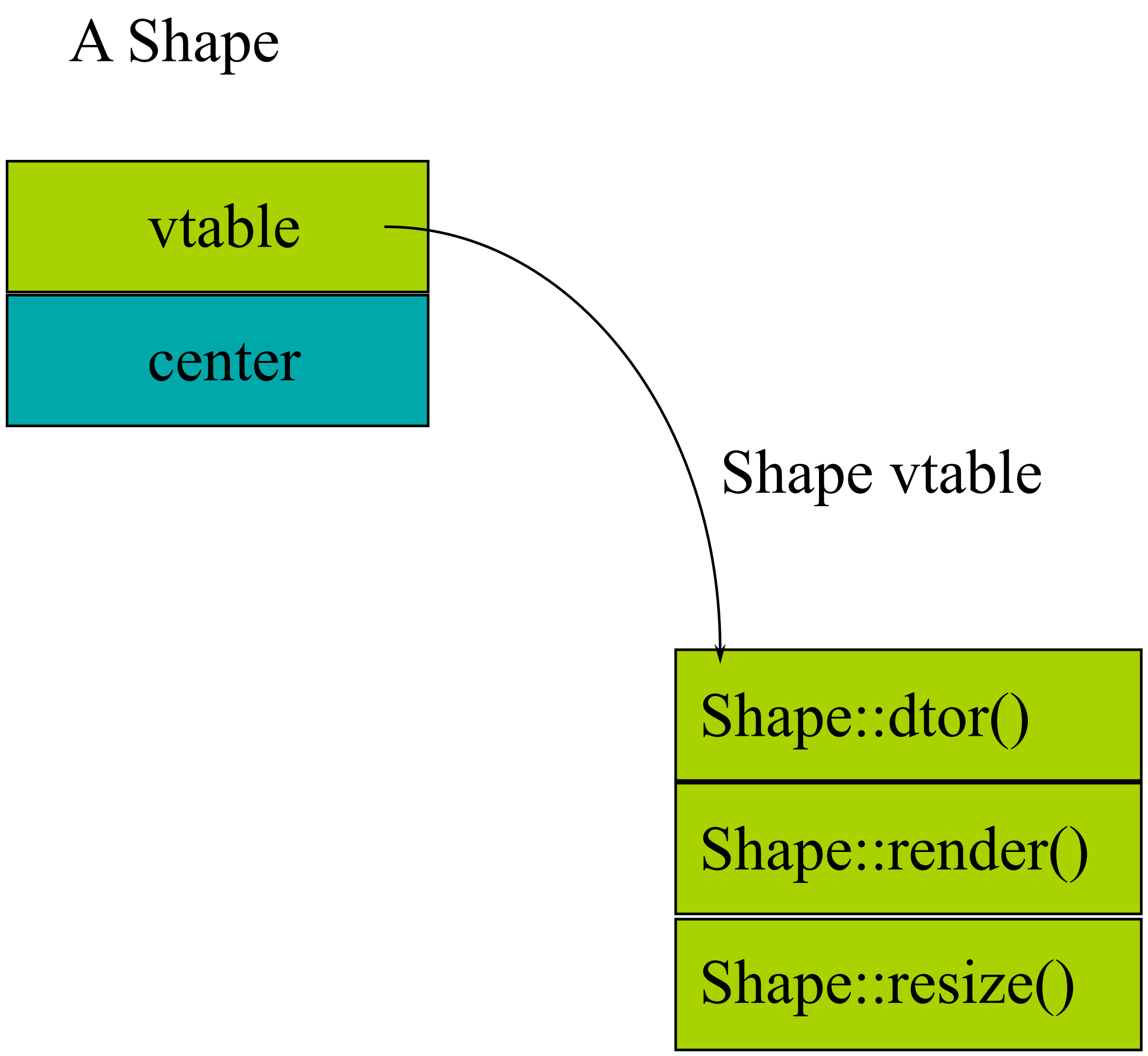
- Non-virtual functions
  - Compiler generates ***static***, or direct call to stated type
  - Faster to execute
- Virtual functions 透明地
  - Can be ***transparently*** overridden in a derived class
  - Objects carry a pack of their virtual functions 一组
  - Compiler checks pack and ***dynamically*** calls the right function
  - If compiler knows the function at compile-time, it can generate a static call

当一个父类中的成员函数声明为虚函数时，它的子类中的同名函数会自动继承虚函数的特性，即使子类在函数声明中没有显式地加上 "virtual" 关键字。

How virtuals work in C++

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

see: virtual.cpp



A Shape

vtable

center

Shape vtable

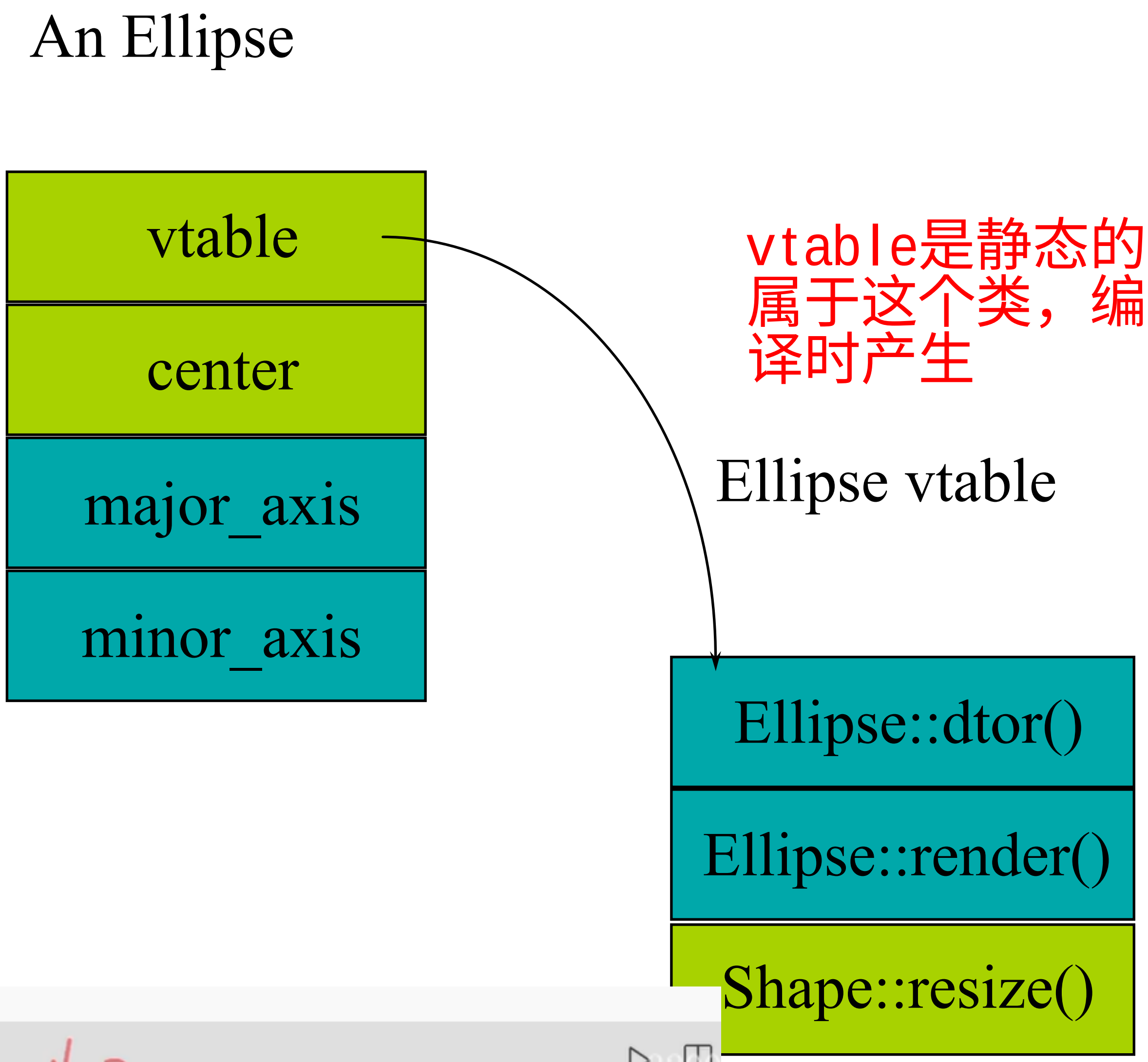
Shape::dtor()

Shape::render()

Shape::resize()

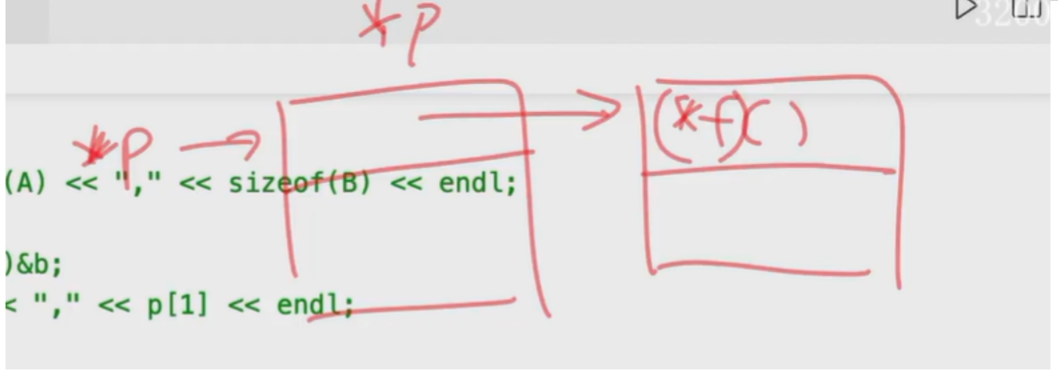
Ellipse

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



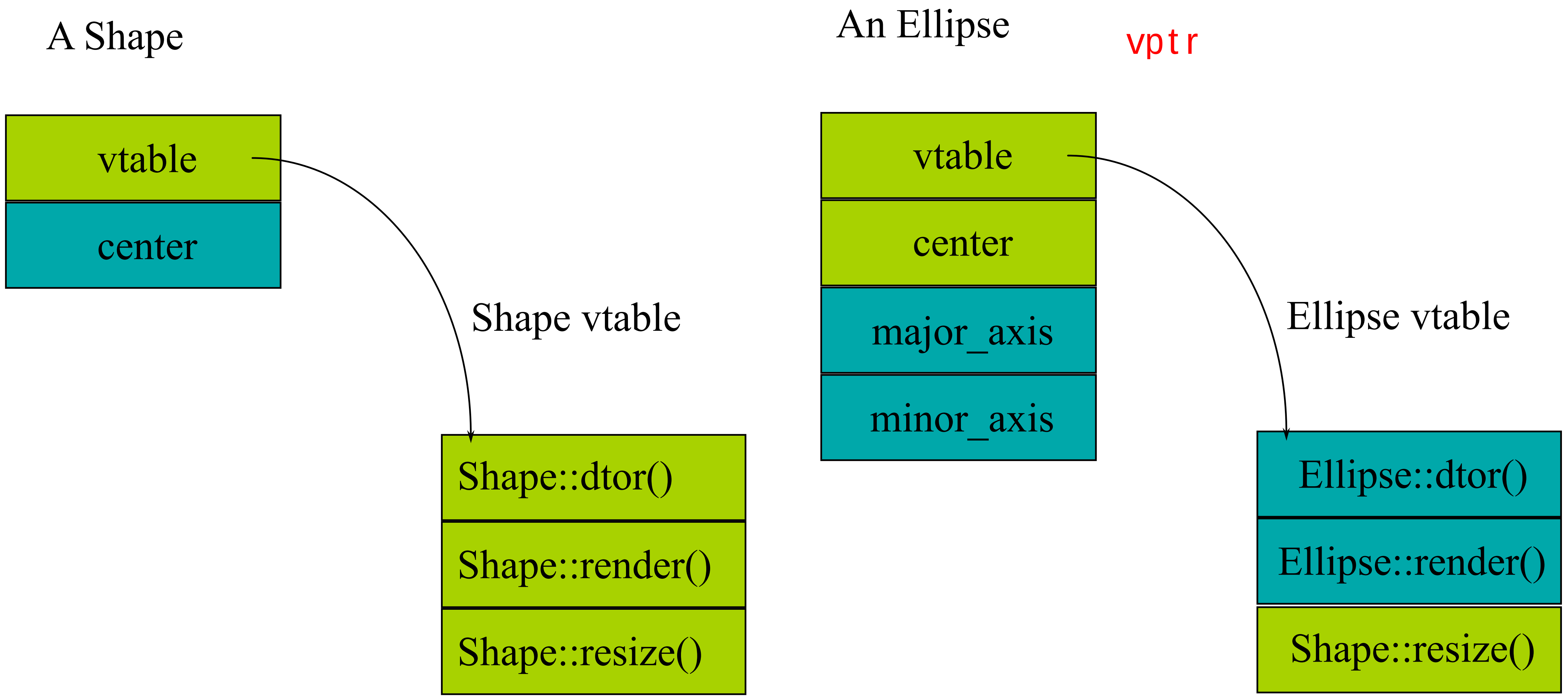
vtable是静态的属于这个类，编译时产生

父类构造函数中调用virtual函数时仍然是父类的
普通函数中调用virtual函数会调用子类的
vttr是在构造时候写入的，调用构造函数时会指向那个类的vtable



```
(A) << "," << sizeof(B) << endl;
)&b;
< "," << p[1] << endl;
```

Shape vs Ellipse



A Shape

vtable

center

Shape vtable

Shape::dtor()

Shape::render()

Shape::resize()

An Ellipse

vptr

vtable

center

major_axis

minor_axis

Ellipse vtable

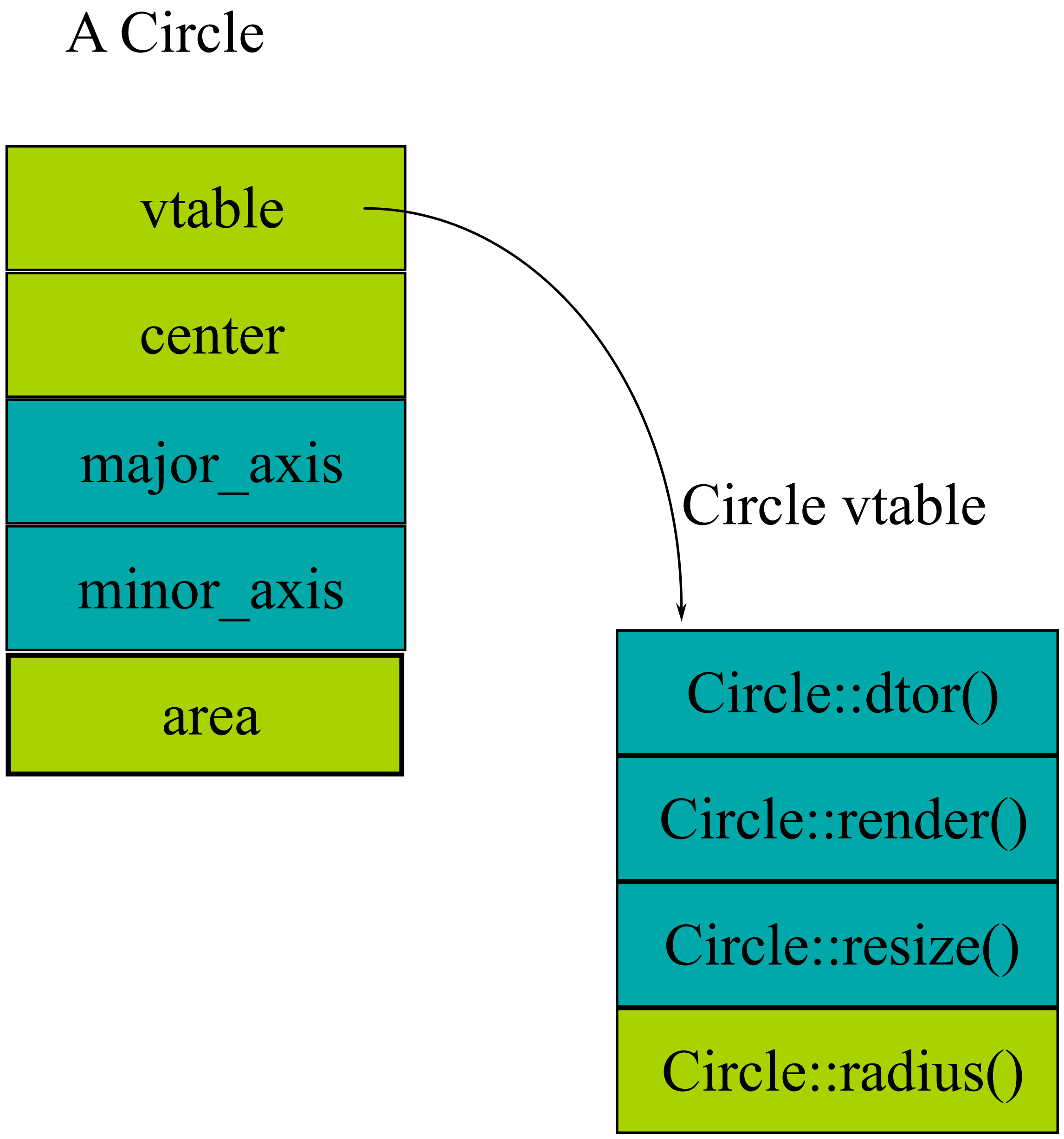
Ellipse::dtor()

Ellipse::render()

Shape::resize()

Circle

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



A Circle

vtable

center

major_axis

minor_axis

area

Circle vtable

Circle::dtor()

Circle::render()

Circle::resize()

Circle::radius()

What happens if

子类对象赋值给父类
切除子类中多出来的

```
Ellipse elly(20F, 40F);
Circle  circ(60F);
elly = circ; // 10 in 5?
```

构造出来后，vptr不会变，所以elly
中的vptr在将circ赋值给elly后还是
ellt原来的，不会被子类的覆盖

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

```
elly.render(); // Ellipse::render()
```

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()
```

前提：render是虚函数
如果没有virtual，就是静态绑定，还是会调用elly的render

Virtuals and reference arguments

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

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;
```

p所指的对象的析构函数会被调用

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

Overriding

- Overriding redefines the body of a virtual function

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

名称，参数完全一致

返回类型可以有继承关系：
比如Base中func返回Base*
子类中返回Derived*

子类重写函数可以返回与父类中相同的类型，或者是父类中返回类型的子类型
子类中返回父类函数返回类型的父类型不行

父类无virtual，子类有virtual，两个函数没关系，子类和孙子才有关系

如果父类中多个重载都virtual，子类只重写一个，其他的会name hide

Calls up the chain

- You can still call the overridden function:

```
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!
};
```

返回类型是对象时，不能有继承关系，因为会发生slice off

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

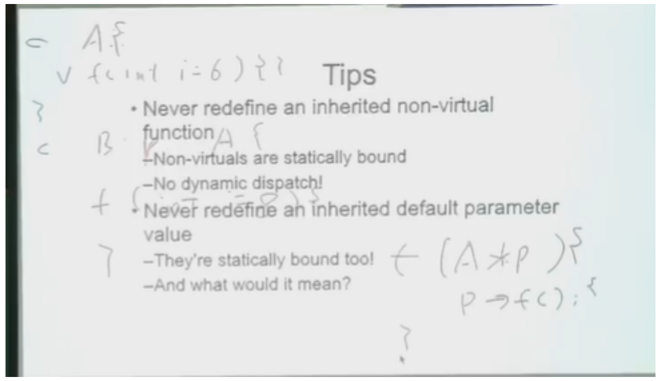
- When you *override* an *overloaded* function, override all of the variants!

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

Tips

不要在子类中重新定义父类中非virtual的韩式，因为会容易让人误解是重写的

- Never redefine an inherited non-virtual function
  - –Non-virtuals are statically bound
  - –No dynamic dispatch! 动态调用
- Never redefine an inherited default parameter value
  - –They’re statically bound too!
  - –And what would it mean?



```
c A{
  v f(int i=6){}
}
c B : A {
  f(int i=8){}
}
f(A*p){
  p->f();
}
```

Tips

- Never redefine an inherited non-virtual function
  - –Non-virtuals are statically bound
  - –No dynamic dispatch!
- Never redefine an inherited default parameter value
  - –They’re statically bound too!
  - –And what would it mean?

i等于6，因为编译器静态绑定时认为p是父类，把i设置为默认值6了

Virtual in Ctor?

构造函数里调用虚函数，还是调用父类的，因为构造时vptr就存在了，不会发生动态绑定

```
class A {
public:
  A() { f(); }
  virtual void f() { cout << “A::f()”; }
};
class B : public A {
public:
  B() { f(); }
  void f() { cout << “B::f()”; }
};
```

Abstract base classes

- An *abstract base class* has pure virtual functions
  - Only interface defined
  - No function body given
- *Abstract base classes cannot be instantiated*
  - Must derive a new class (or classes)
  - Must supply definitions for all pure virtuals before class can be instantiated

提供

In C++

- Define the general properties of a Shape

```
class XYPos{ ... };    // x,y point
class Shape {
public:
  Shape();
  virtual void render() = 0; // mark
  render() pure
  void move(const XYPos&);
  virtual void resize();
protected:
  XYPos center;
};
```

一旦类内有一个纯虚函数，这个类就不能被用来实例化

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

一个类没有任何非静态成员变量，不管是继承过来的还是自身定义的成员变量。

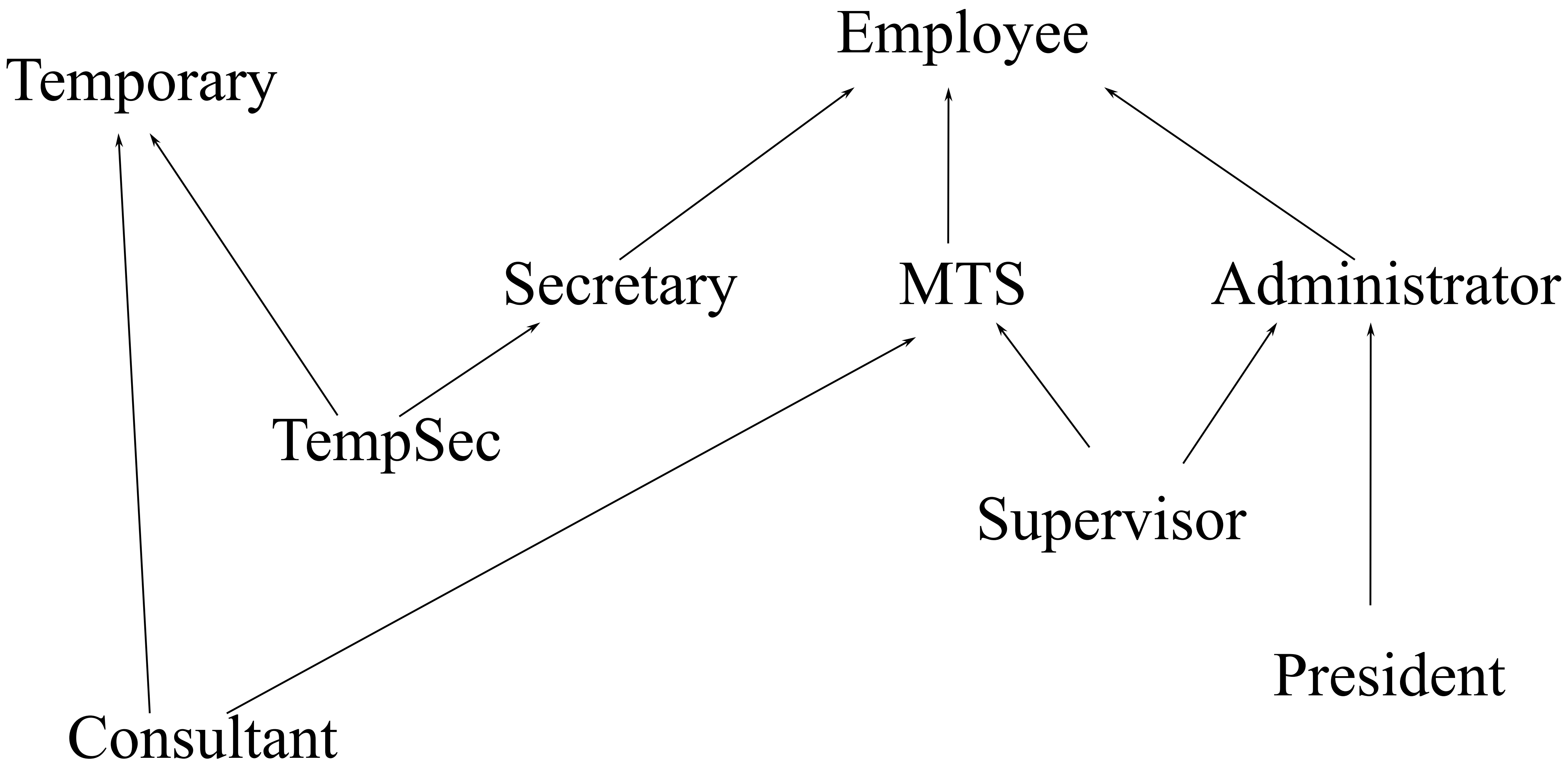
静态成员变量相当于全局变量

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;
};
```

Multiple Inheritance



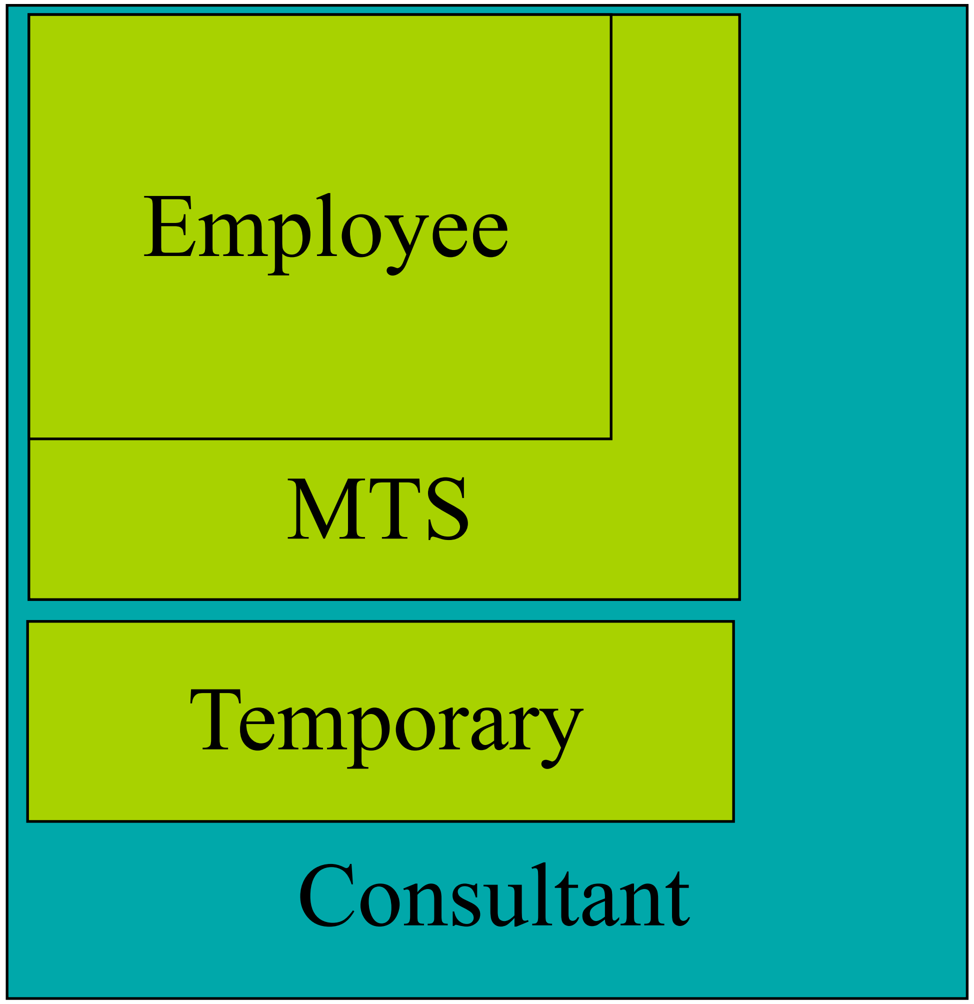
Mix and match

```
class Employee {
protected:
String name;
EmpID id;
};

class MTS : public Employee {
protected:
Degrees degree_info;
};

class Temporary {
protected:
Company employer;
};
```

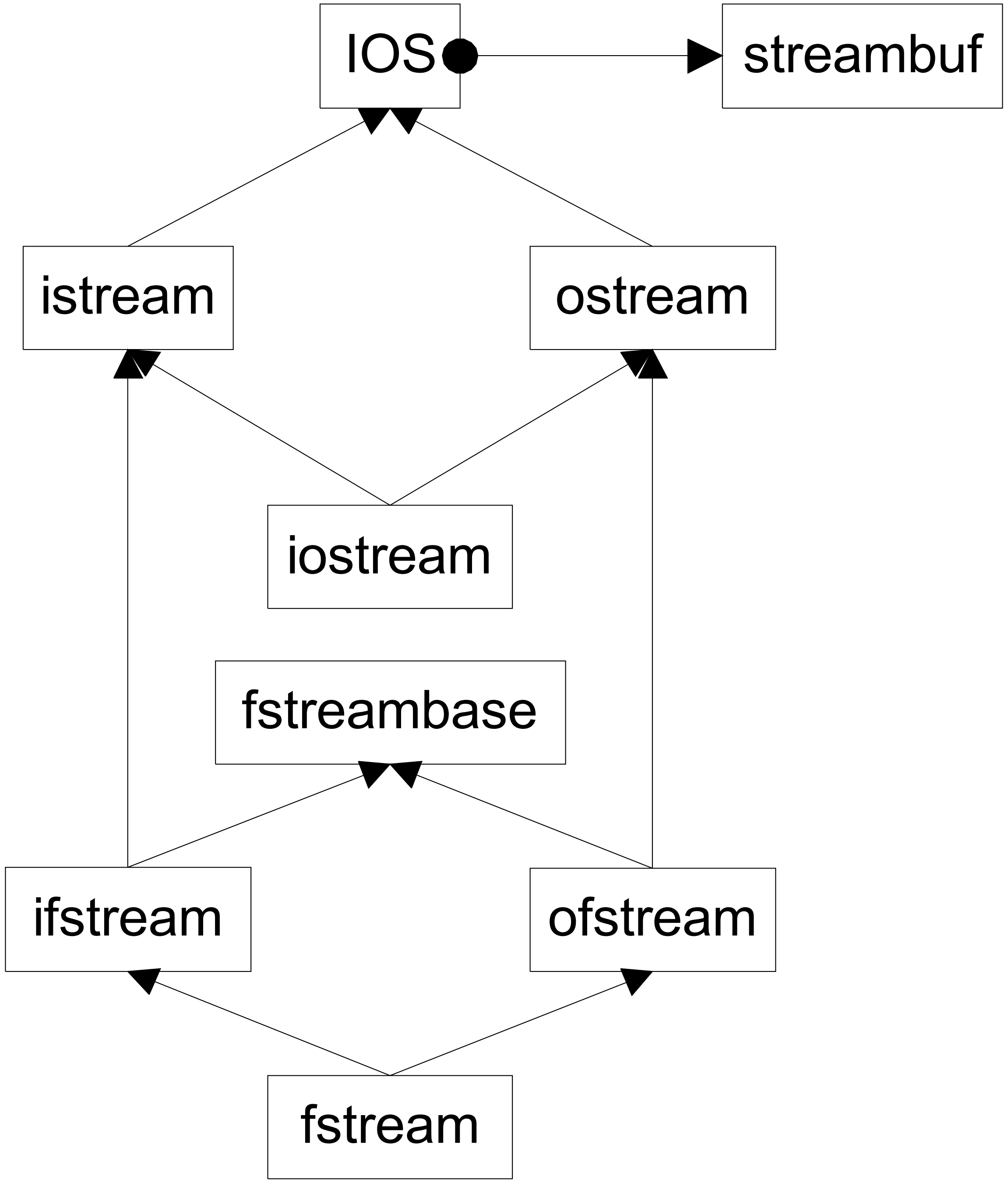
```
class Consultant:
   public MTS,
   public Temporary {

...
};
```

- Consultant picks up the attributes of both MTS and Temporary.
  - name
  - id
  - employer
  - degree_info

MI Complicates Data Layouts



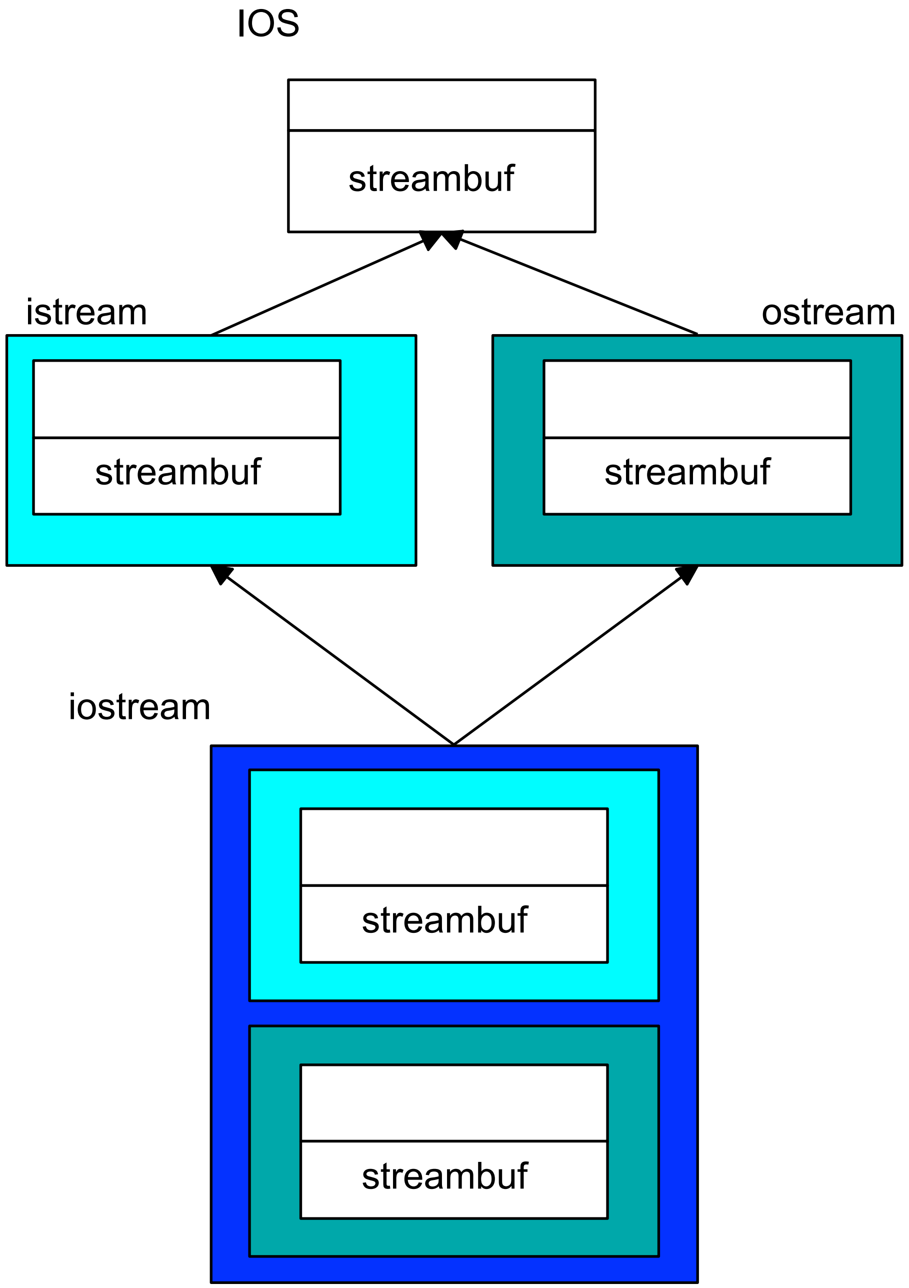
IOStreams package

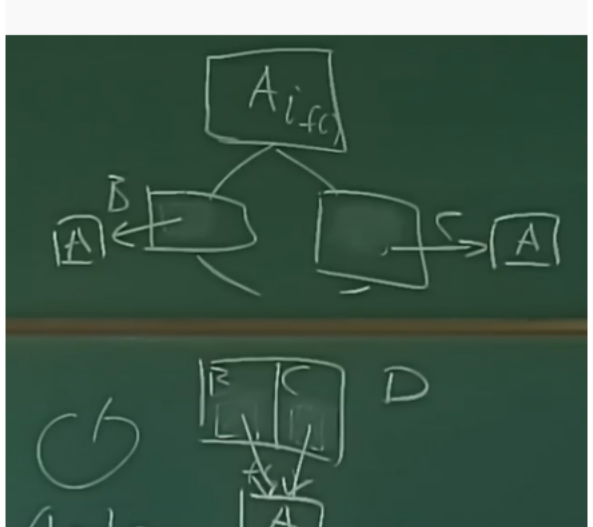


Vanilla MI

“普通的（原始的）多重继承”
（Vanilla Multiple Inheritance
）

- Members are duplicated
- Derived class has access to full copies of each base class
- This ***can*** be useful!
  - –Multiple links for lists
  - –Multiple streambufs for input and output





虚继承

More on MI...

```
class B1 { int m_i; };
class D1 : public B1 {};
class D2 : public B1 {};
class M : public D1, public D2 {};

void main() {
  M m;  // OK
  B1* p = new M; // ERROR: which B1
  B1* p2 = dynamic_cast<D1*>(new M); // OK
}
```

B1 is a *replicated* sub-object of M.

Replicated bases

- Normally replicated bases aren’t a problem (usage of B1 by D1 and D2 is an implementation detail).
- Replication becomes a problem if replicated data makes for confusing logic:

```
M m;
m.m_i++; // ERROR: D1::B1.m_i or
D2::B1.m_i?
```

Safe uses

- Protocol classes

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;
};
```

Safe uses

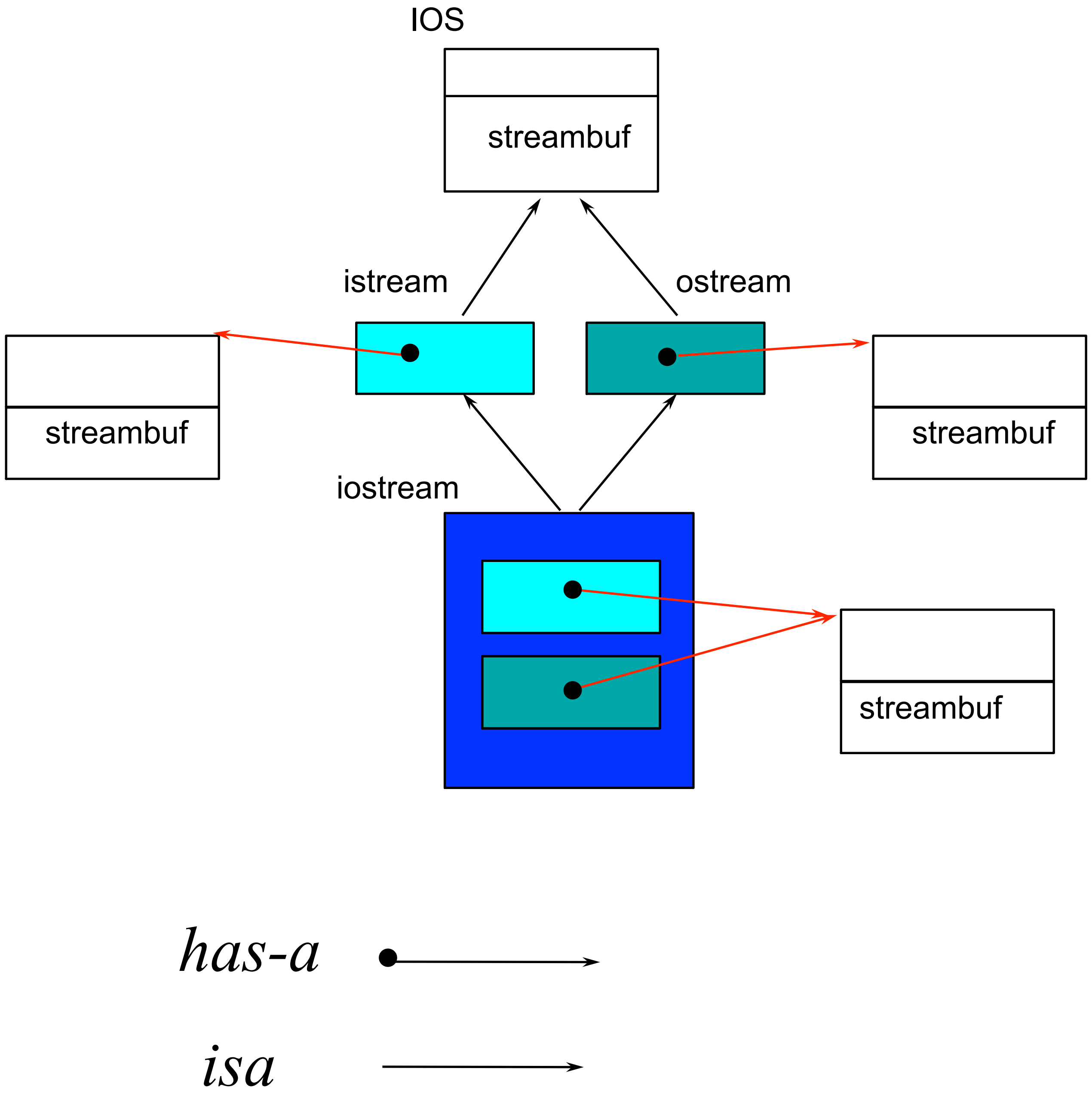
- Protocol classes

What about sharing?

- How do you avoid having two streambufs?
- Base classes can be ***virtual***
  - To C++ people, “virtual” means “indirect”
- Virtual member functions have dynamic binding
  - They use pointer indirection
- Virtual base classes are represented indirectly
  - They use pointer indirection

Using virtual base classes

- Virtual base classes are ***shared***
- Derived classes have a single copy of the virtual base
- Full control over sharing
  - –Up to you to choose
- Cost is in complications

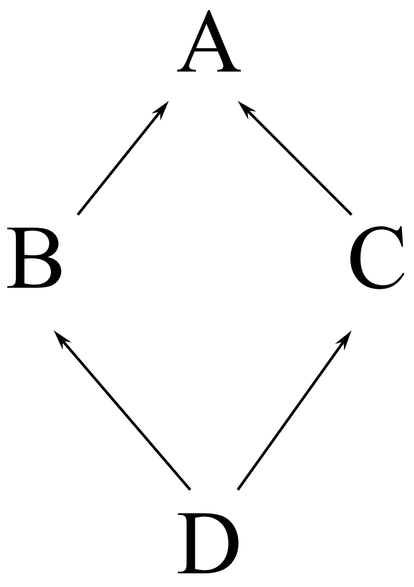


Virtual bases

```
class B1 { int m_i; };
class D1 : virtual public B1 {};
class D2 : virtual public B1 {};
class M : public D1, public D2 {};

void main() {
   M m;  // OK
   m.m_i++; // OK, there is only one B1 in m.
   B1* p = new M; // OK
}
```

Complications of MI

- Name conflicts
  - Dominance rule
- Order of construction
  - Who constructs virtual base?
- Virtual bases not declared when you need them



- Code in virtual bases called more than once
- Compilers are still iffy
- Moral:
  - Use sparingly
  - Avoid diamond patterns
    - expensive
    - hard

Virtual bases

- Use of virtual base imposes some runtime and space overhead.
- If replication isn’t a problem then you don’t need to make bases virtual.
- Abstract base classes (that hold no data except for a vptr) can be replicated with no problem - virtual base can be eliminated.

TIPS for MI

- SAY

NO