Object－Oriented Programming

Week 8

# Polymorphism

Weng Kai

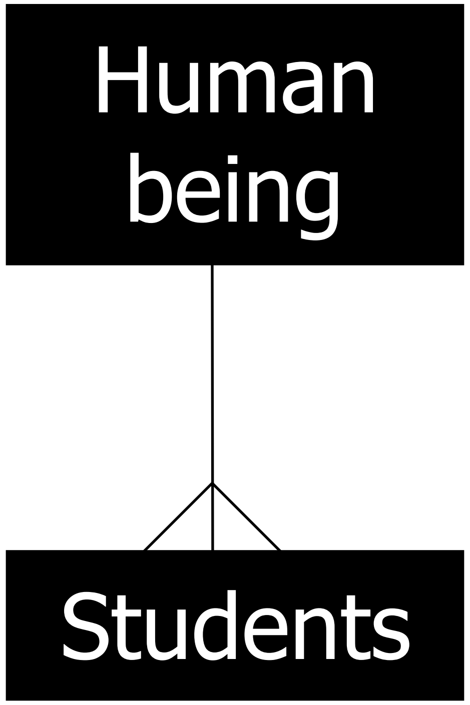
# Conversions

- 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 | $\Rightarrow$ | B |
| D* | $\Rightarrow$ | B* |
| D& | $\Rightarrow$ | B& |

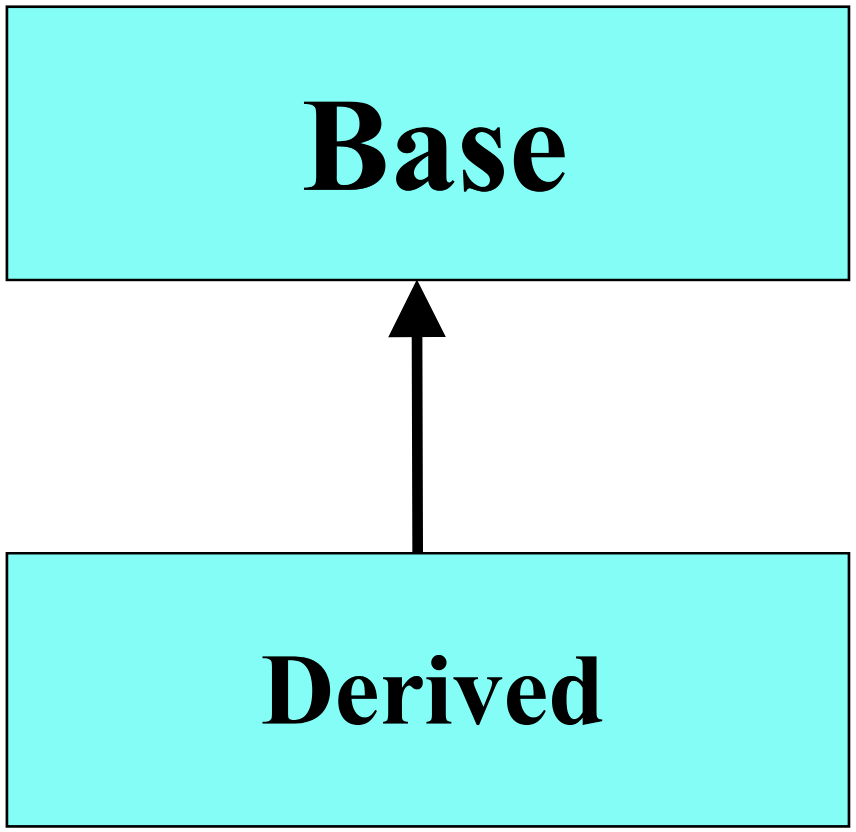
# Up-casting

- 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.



# Upcasting

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



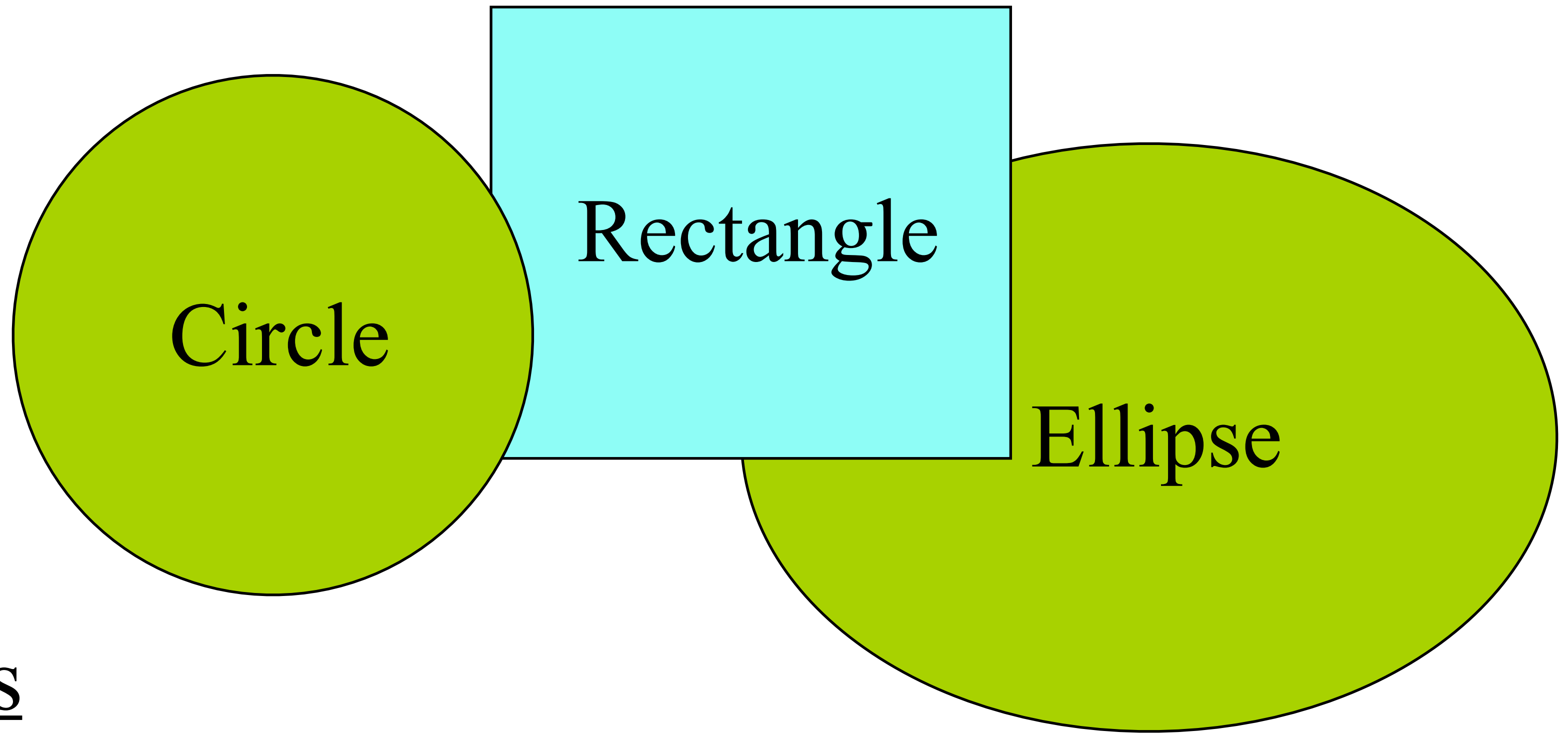
# Upcasting examples

```
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
```

# A drawing program




<u>Operations</u>

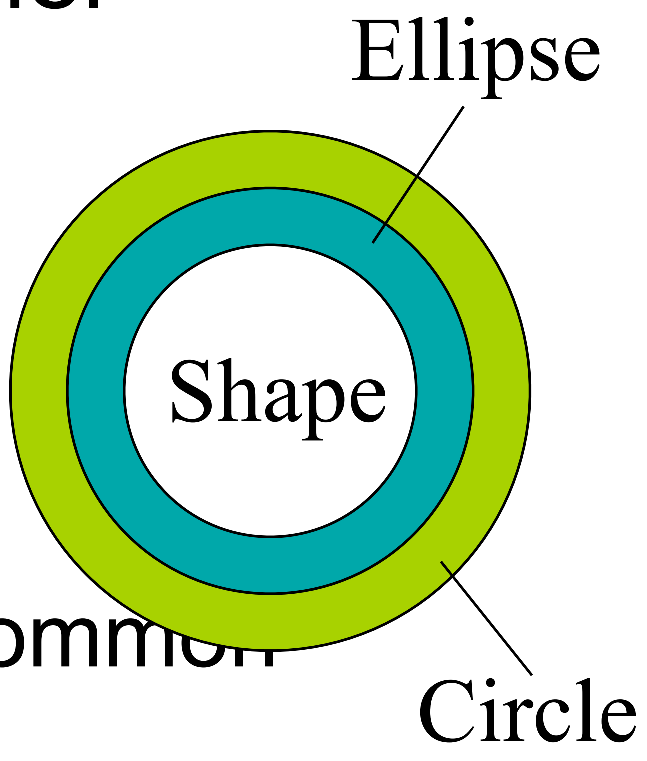
- render
- move
- resize

<u>Data</u>

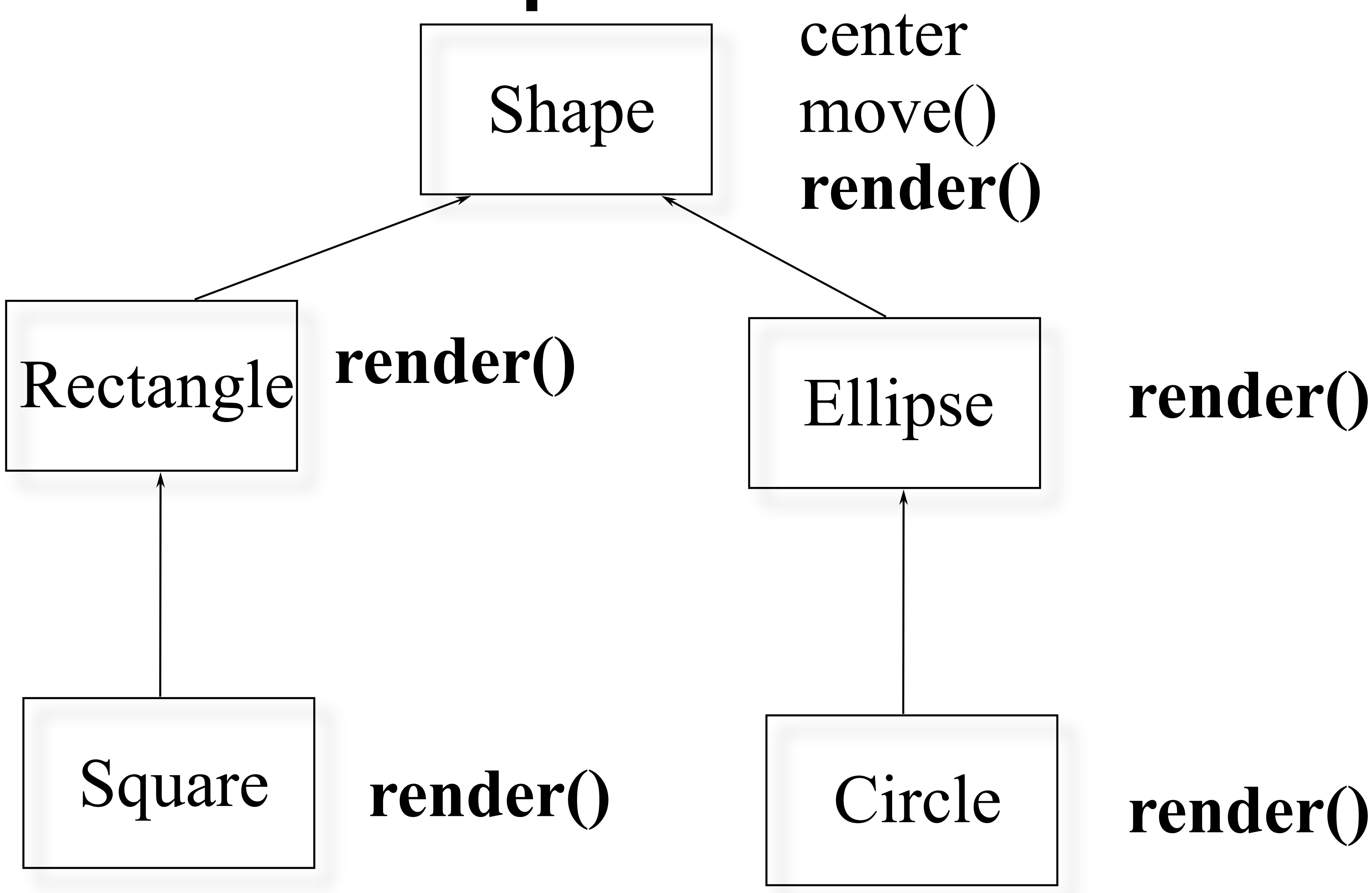
+ center

# 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!*

# 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;
};
```

# 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();
};
```

# 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

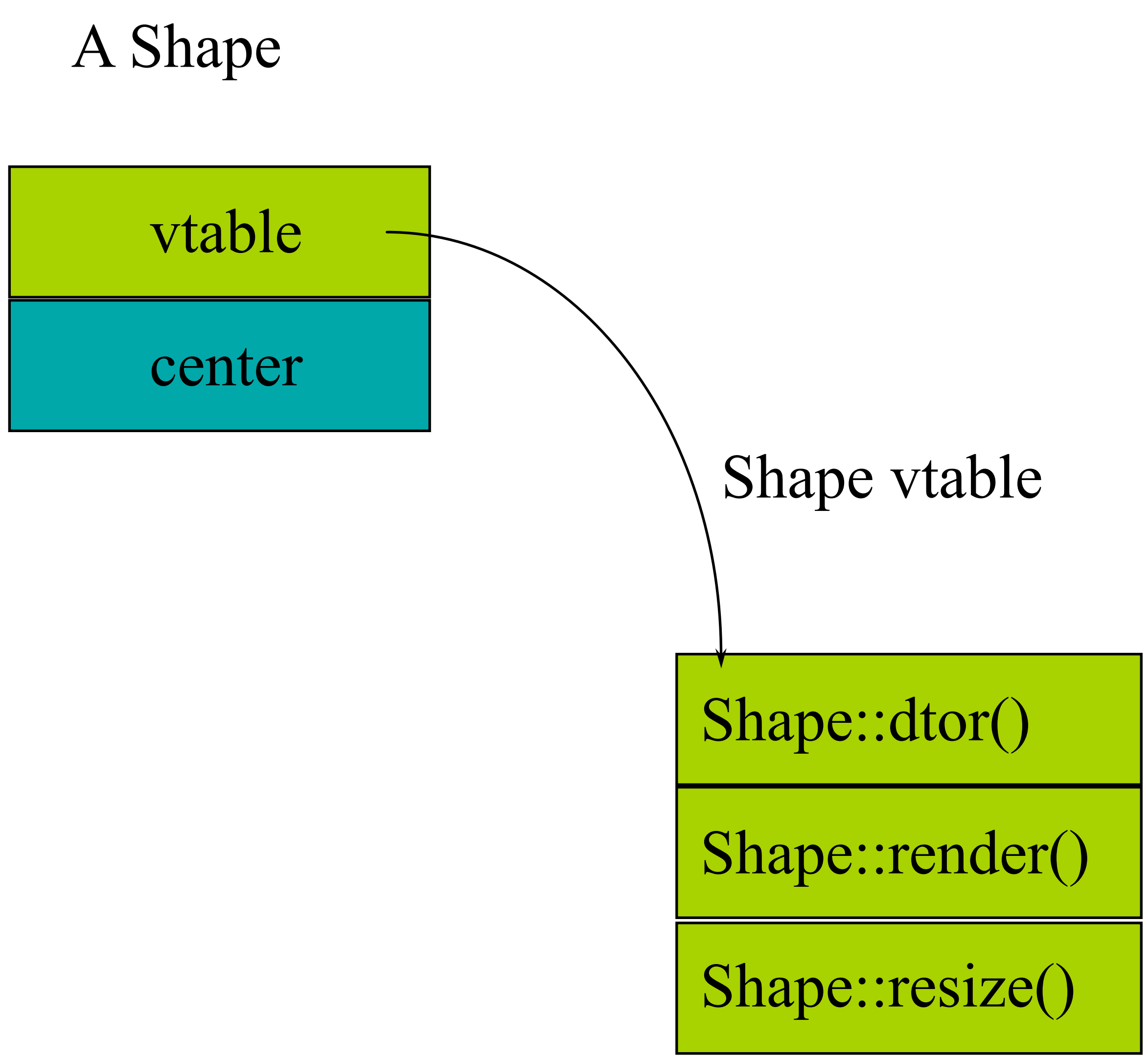
# Virtual functions

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

# How virtuals work in C++

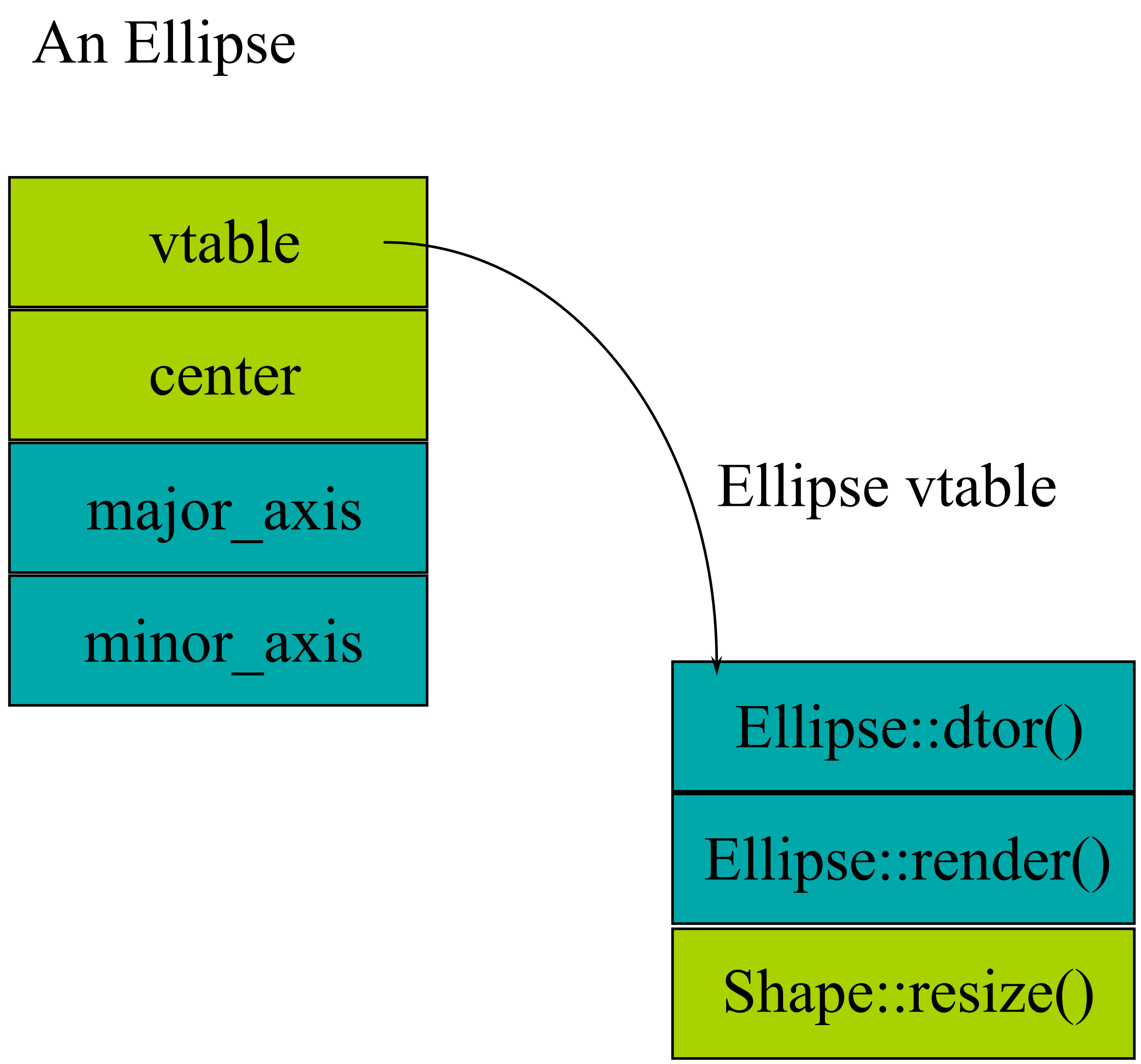
```
class Shape {
public:
   Shape();
   virtual ~Shape();
   virtual void render();
   void move(const
   XYPos&);
   virtual void resize();
protected:
   XYPos center;
};
```

see: virtual.cpp

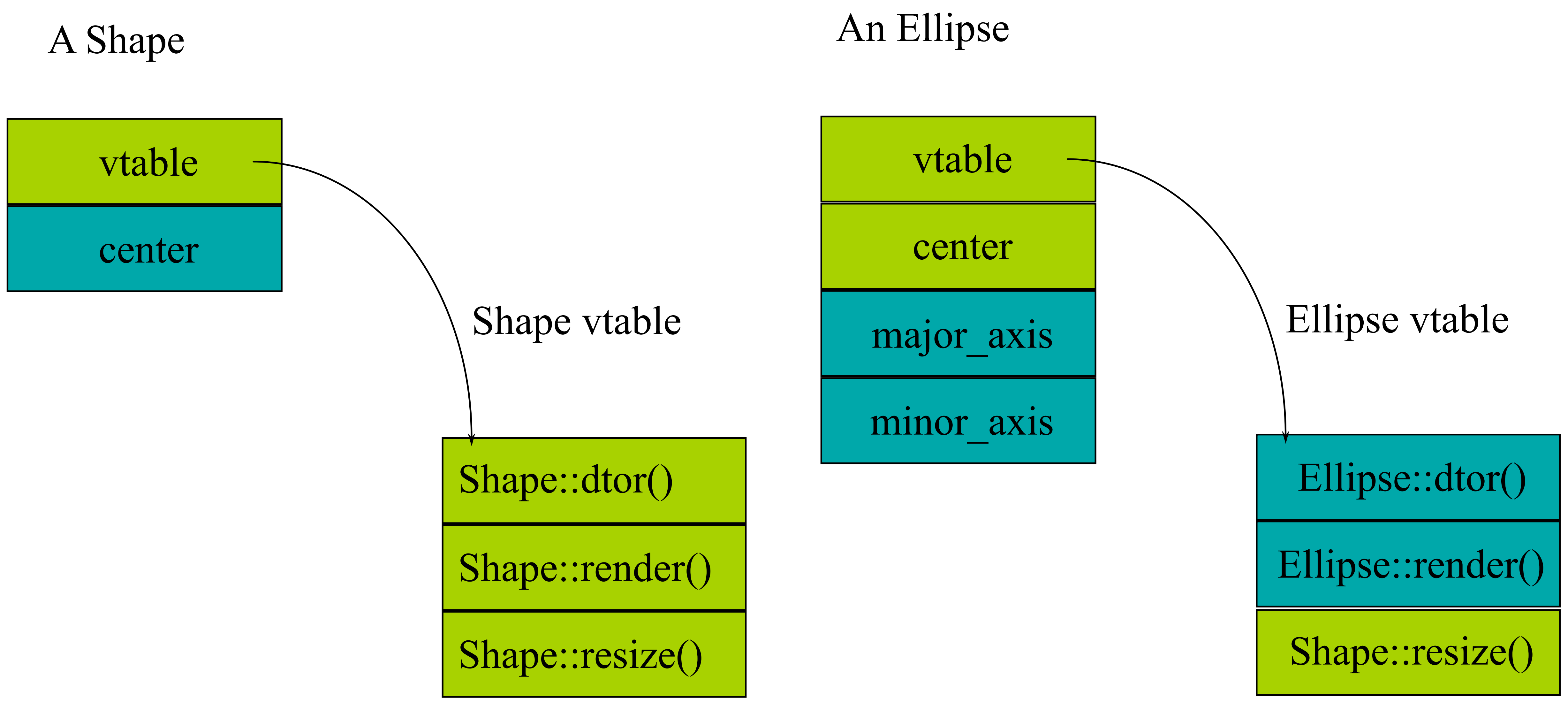


# Ellipse

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

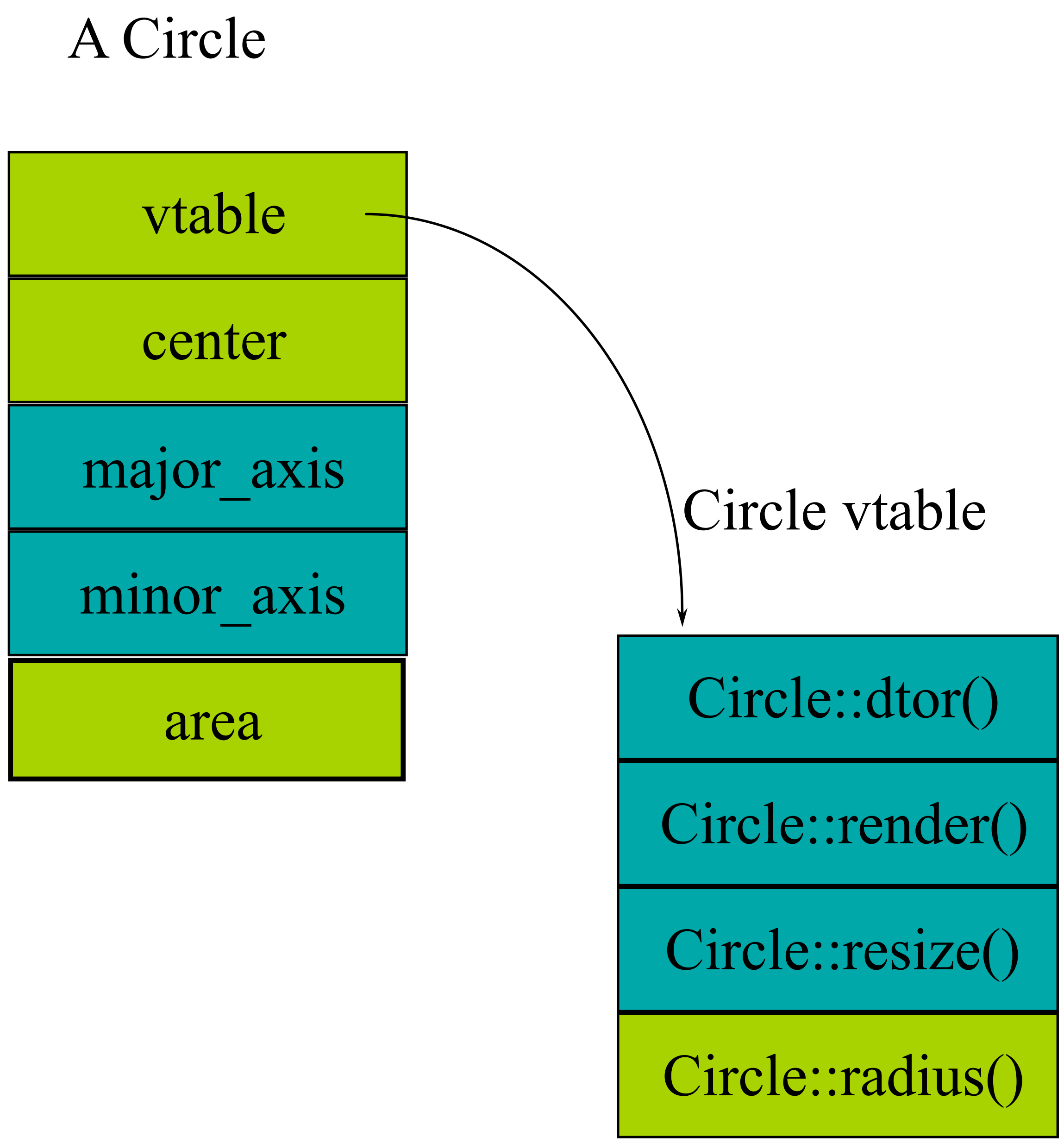


# Shape vs Ellipse



# Circle

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



# What happens if

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

- 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()
```

# Virtuals 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;
```

- 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()
}
```

# 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!
};
```

# 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

- 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?

# Virtual in Ctor?

```
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

# 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;
};
```

# 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

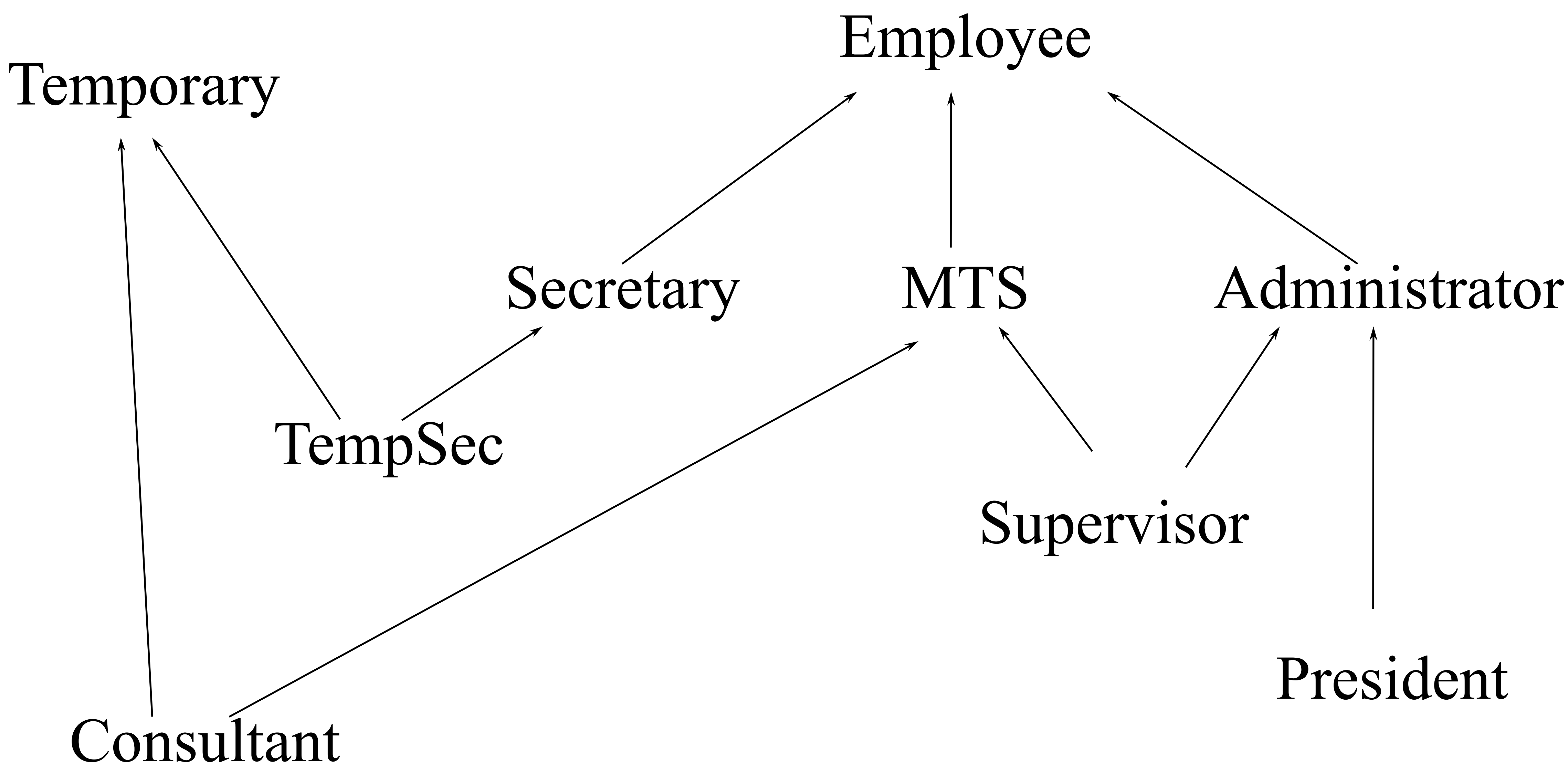
- Abstract base class with
  - All non-static member functions are *pure* virtual except destructor
  - Virtual destructor with empty body
  - No non-static member variables, inherited or otherwise
    - May contain static members

# 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;
};
```

# Multiple Inheritance



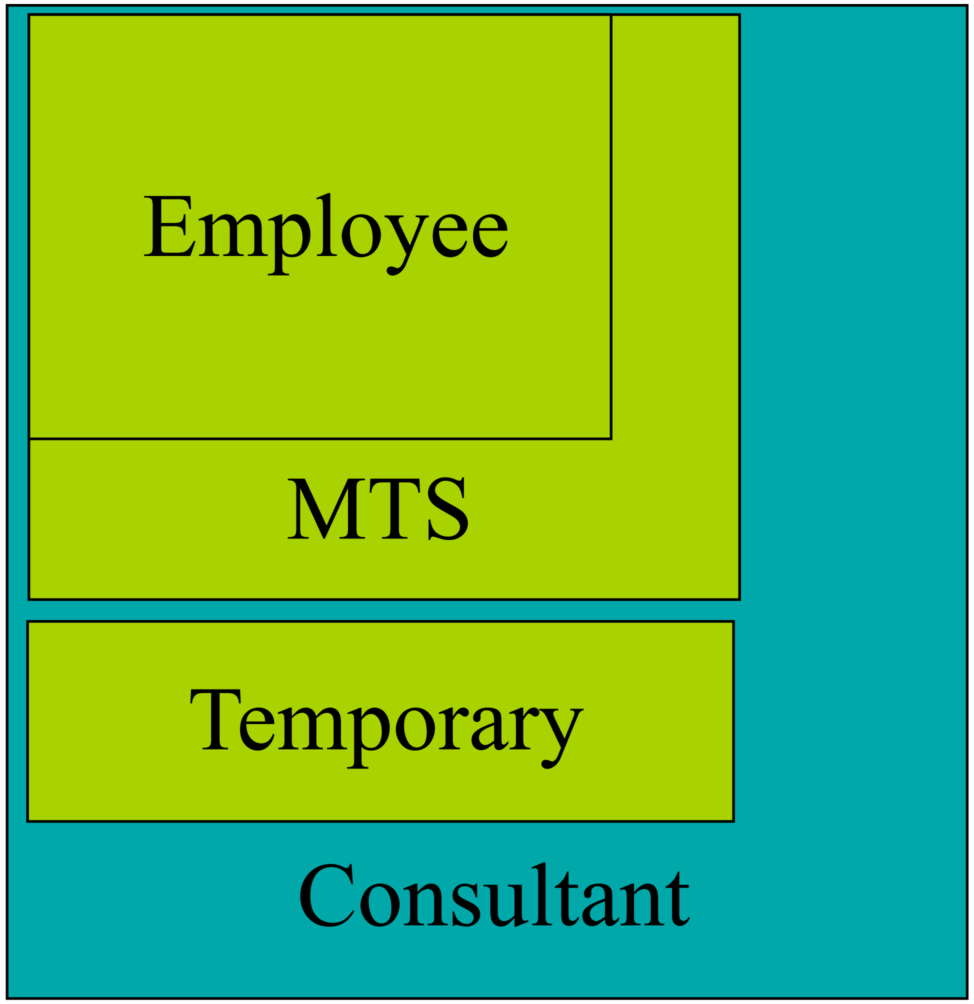
# Mix and match

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

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

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

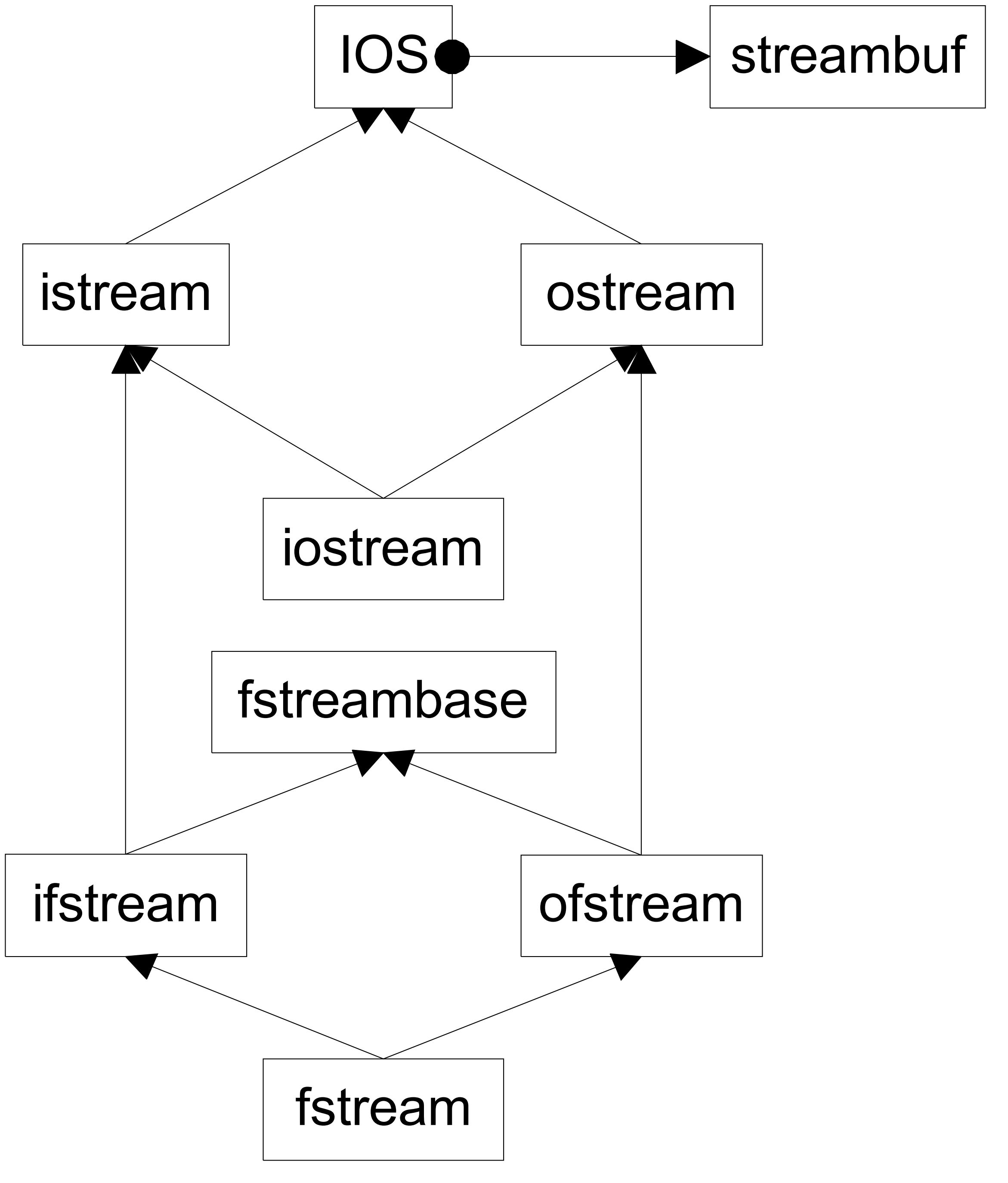
```
class Consultant:
   public MTS,
   public Temporary {

...
};
```

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

# MI Complicates Data Layouts

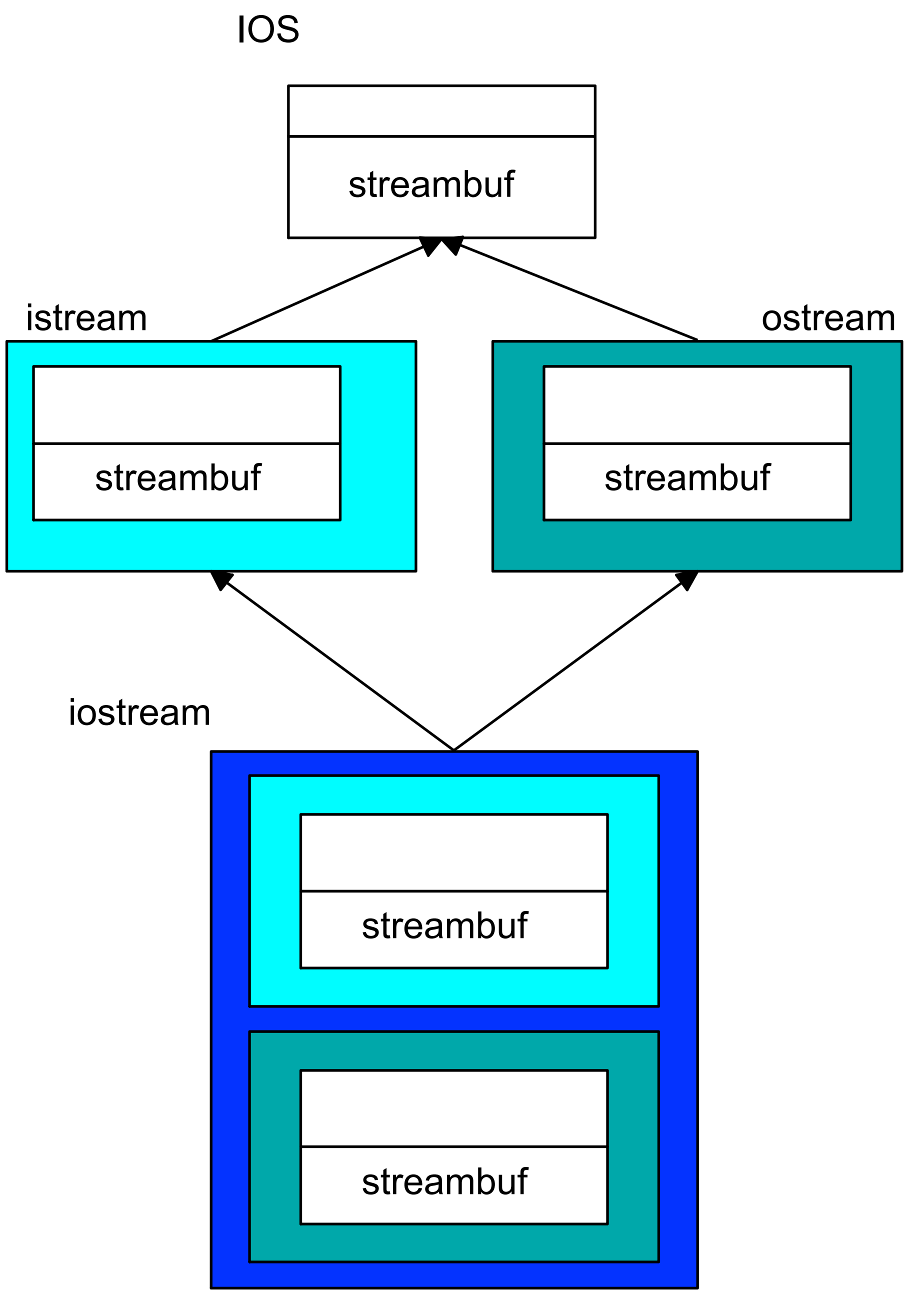


# IOStreams package



# Vanilla MI

- 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



# 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

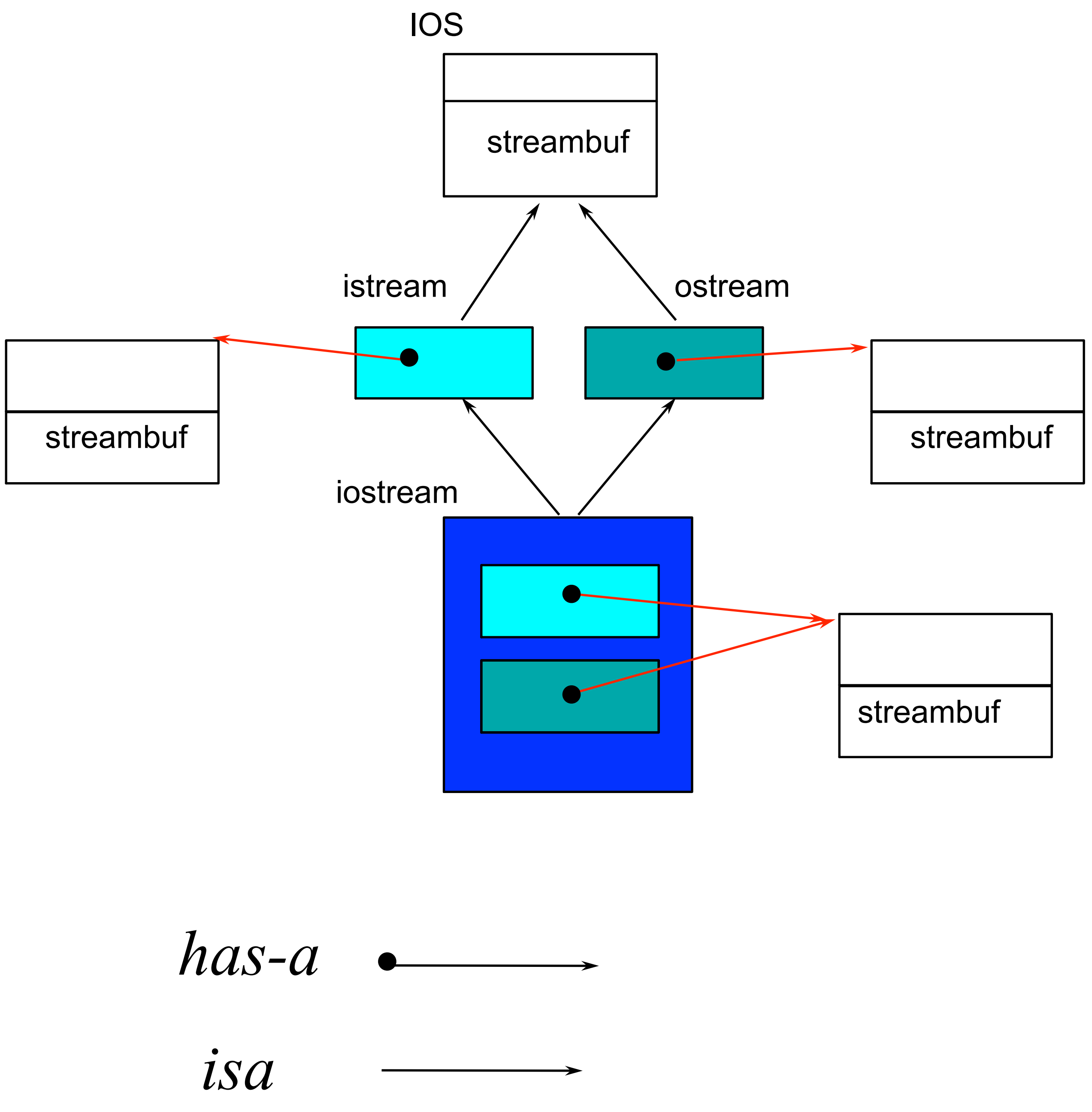
- Protocol classes

# 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

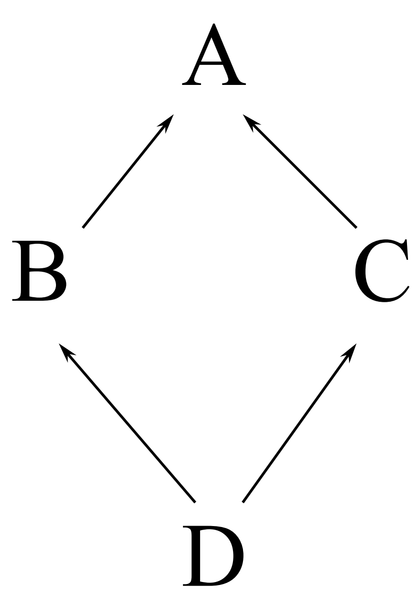


# Virtual bases

```
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