Object – Oriented Programming

Week 11, Spring 2009

Inheritance

Weng Kai

http://fm.zju.edu.cn

Wednesday, 6 May, 2009

Composition

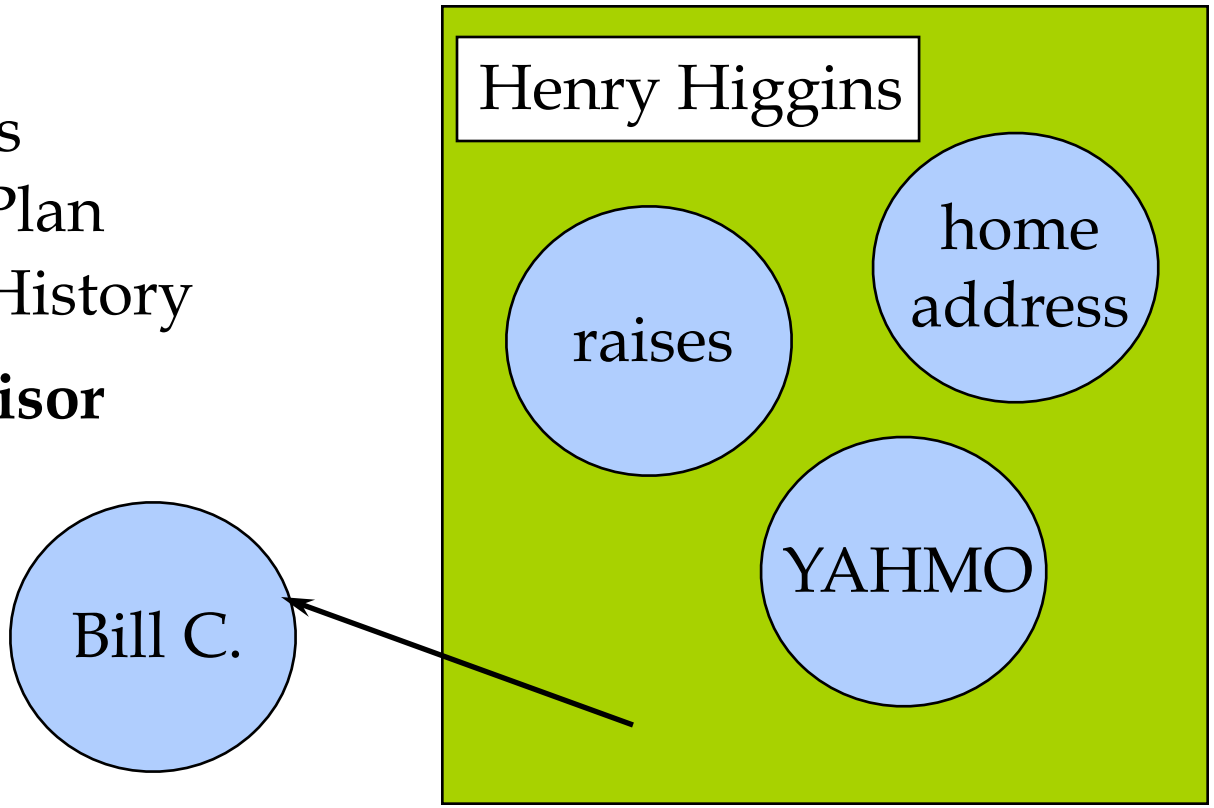
- Objects can be used to build up other objects
- Ways of inclusion
  - Fully
  - By reference
- Inclusion by reference allows sharing
- For example, an Employee has a
  - Name
  - Address
  - Health Plan
  - Salary History
    - Collection of Raise objects
  - Supervisor
    - Another Employee object!

Composition in action

Classes

Employe
e
- Name
- Address
- HealthPlan
- Salary History
- **Supervisor**

Instances



Example

```
class Person { ... };
class Currency { ... };
class SavingsAccount {
public:
   SavingsAccount( const char* name,
                   const char* address, int cents );
   ~SavingsAccount();
   void print();
private:
   Person m_saver;
   Currency m_balance;
};
```

Example...

```
SavingsAccount::SavingsAccount ( const
  char* name, const char* address,
  int cents ) : m_saver(name, address),
  m_balance(0, cents) {}

void SavingsAccount::print() {
   m_saver.print();
   m_balance.print();
}
```

Embedded objects

- All embedded objects are initialized
  - The default constructor is called if
    - you don’t supply the arguments, and there is a default constructor (or one can be built)
- Constructors can have initialization list
  - any number of objects separated by commas
  - is optional
  - Provide arguments to sub-constructors
- Syntax:

```
name( args ) [‘:’ init-list] ‘{‘
```

Question

- If we wrote the constructor as (assuming we have the set accessors for the subobjects):

```
SavingsAccount::SavingsAccount ( const char* name,
   const char* address, int cents ) {
      m_saver.set_name( name );
      m_saver.set_address( address );
      m_balance.set_cents( cents );
}
```

- Default constructors would be called

Public vs. Private

- It is common to make embedded objects private:
  - they are part of the underlying implementation
  - the new class only has part of the public interface of the old class
- Can embed as a public object if you want to have the entire public interface of the subobject available in the new object:

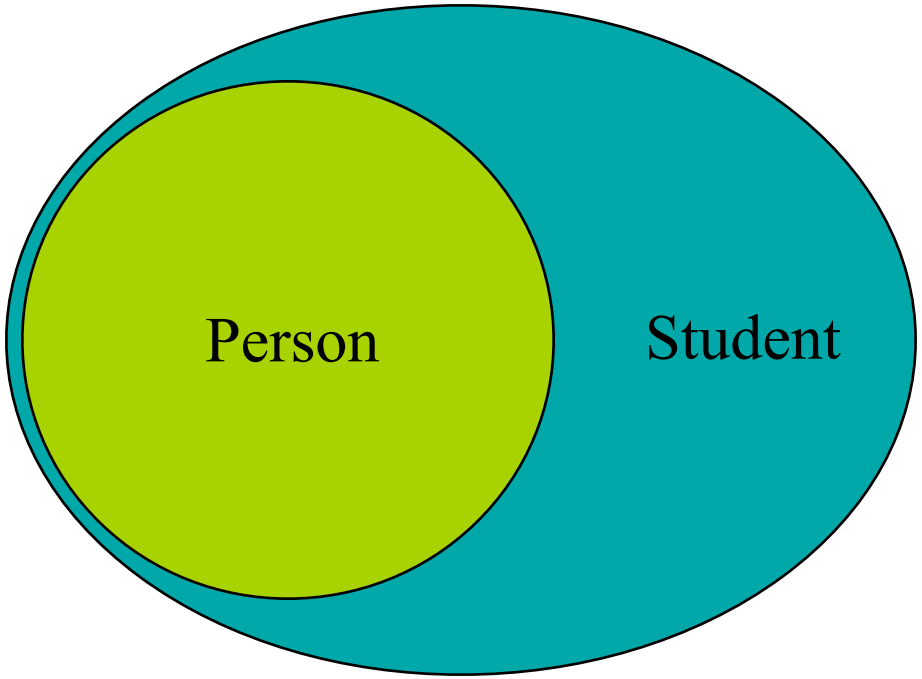
```
class SavingsAccount {
 public:
          Person m_saver;  ... };     // assume
  Person class has set_name()
SavingsAccount  account;
account.m_saver.set_name("Fred" );
```

Inheritance

- Language implementation technique
- Also an important component of the OO design methodology
- Allows sharing of
  - Member data
  - Member functions
  - Interfaces
- Key technology in C++

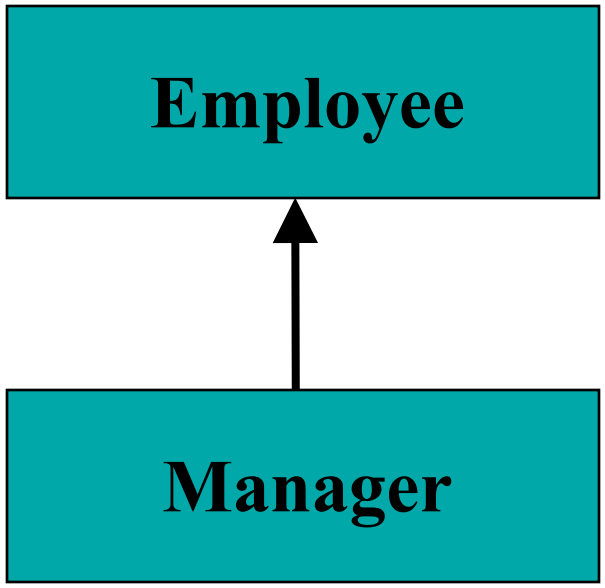
Inheritance

- The ability to define the behavior or implementation of one class as a ***superset*** of another class



Inheritance

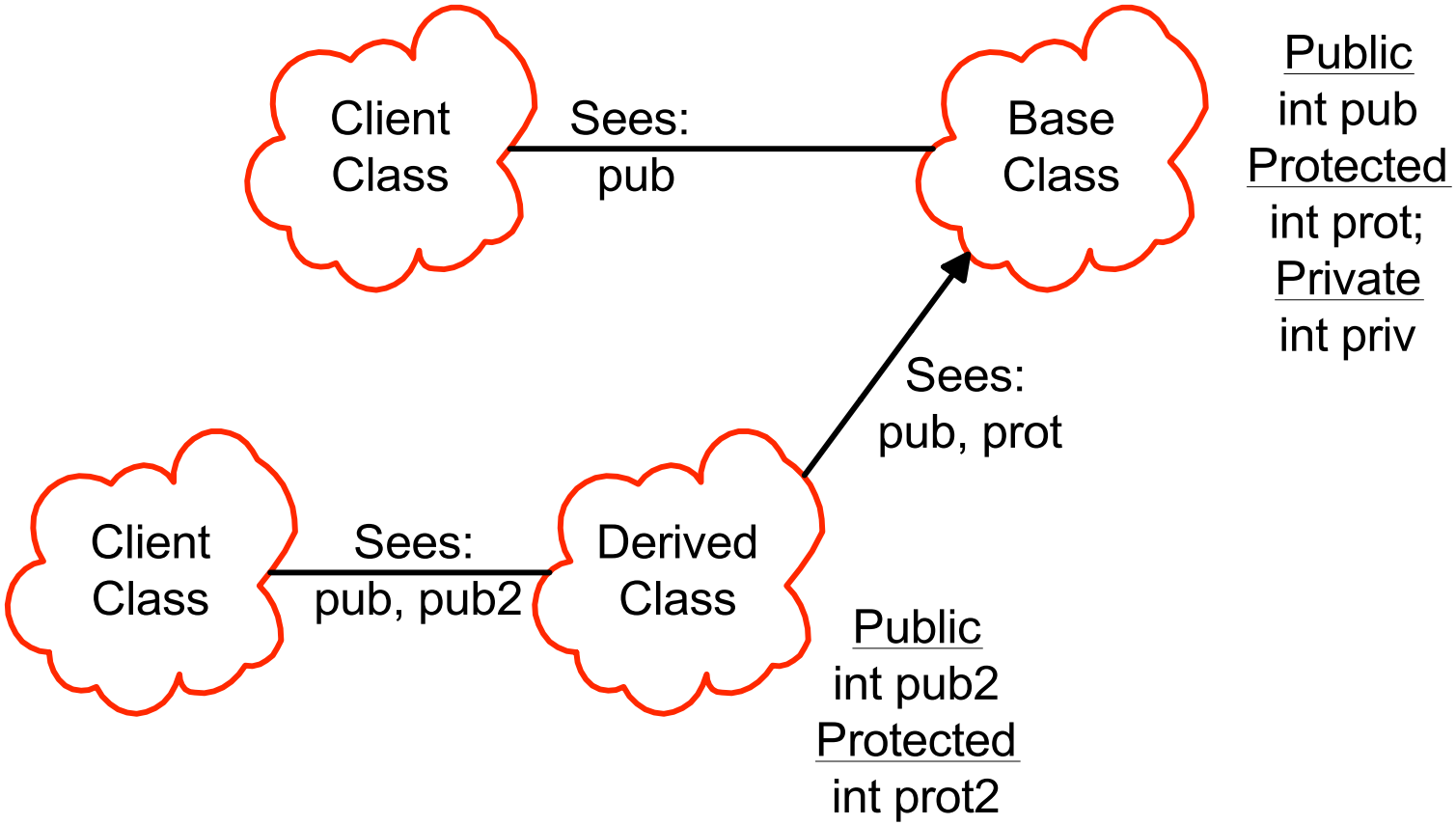
- **Class relationship: Is-A**



Base Class
Super
Parent

Derived Class
Sub
Child

Scopes and access in C++



Declare an Employee class

```
class Employee {
public:
   Employee( const std::string& name,
  const std::string& ssn );

   const std::string& get_name() const;
   void print(std::ostream& out) const;
   void print(std::ostream& out, const
  std::string& msg) const;
protected:
   std::string m_name;
   std::string m_ssn;
};
```

Constructor for Employee

```
Employee::Employee( const string& name,
                    const string& ssn )
   : m_name(name), m_ssn( ssn)
{
    // initializer list sets up the values!
}
```

Employee member functions

```
inline const std::string& Employee::get_name() const
   {
    return m_name;
}
inline void Employee::print( std::ostream& out )
   const {
    out << m_name << endl;
    out << m_ssn << endl;
}
inline void Employee::print(std::ostream& out, const
   std::string& msg) const {
    out << msg << endl;
    print(out);
}
```

Now add Manager

```
class Manager : public Employee {
public:
    Manager(const std::string& name,
                    const std::string& ssn,
  const std::string& title);
    const std::string title_name() const;
    const std::string& get_title() const;
    void print(std::ostream& out) const;
private:
    std::string m_title;
};
```

Inheritance and constructors

- Think of inherited traits as an embedded object

- Base class is mentioned by class name

```
Manager::Manager( const string& name, const string&
ssn, const string& title = "" )
   :Employee(name, ssn), m_title( title )
{
}
```

More on constructors

- Base class is always constructed first
- If no explicit arguments are passed to base class
  - Default constructor will be called
- Destructors are called in exactly the reverse order of the constructors.

Manager member functions

```
inline void Manager::print( std::ostream& out )
   const {
      Employee::print( out );      // call the base
   class print
      out << m_title << endl;
}


inline const std::string& Manager::get_title() const
   {
    return m_title;
}


inline const std::string Manager::title_name() const
   {
    return string( m_title + ": " + m_name );  //
   access base m_name
}
```

Uses

```
int main () {
   Employee bob( "Bob Jones", "555-44-0000" );
   Manager bill( "Bill Smith", "666-55-1234", "Important
   Person" );

   string name = bill.get_name();   // okay Manager
   inherits Employee
   //string title = bob.get_title();    // Error -- bob is
   an Employee!
   cout << bill.title_name() << '\n' << endl;
   bill.print(cout);
   bob.print(cout);
   bob.print(cout, "Employee:");
   //bill.print(cout, "Employee:");   // Error hidden!
}
```

Name Hiding

- If you redefine a member function in the derived class, all other overloaded functions in the base class are inaccessible.
- We'll see how the keyword virtual affects function overloading next time.

What is not inherited?

- Constructors
  - synthesized constructors use memberwise initialization
  - In explicit copy ctor, explicity call base-class copy ctor or the default ctor will be called instead.
- Destructors
- Assignment operation
  - synthesized operator= uses memberwise assignment
  - explicit operator= be sure to explicity call the base class version of operator=
- Private data is hidden, but still present

Access protection

- Members
  - Public: visible to all clients
  - Protected: visible to classes derived from self (and to friends)
  - Private: visible only to self and to friends!
- Inheritance
  - Public: `class Derived : public Base ...`
  - Protected: `class Derived : protected Base ...`
  - Private: `class Derived : private Base ...`
    - `default`

How inheritance affects access

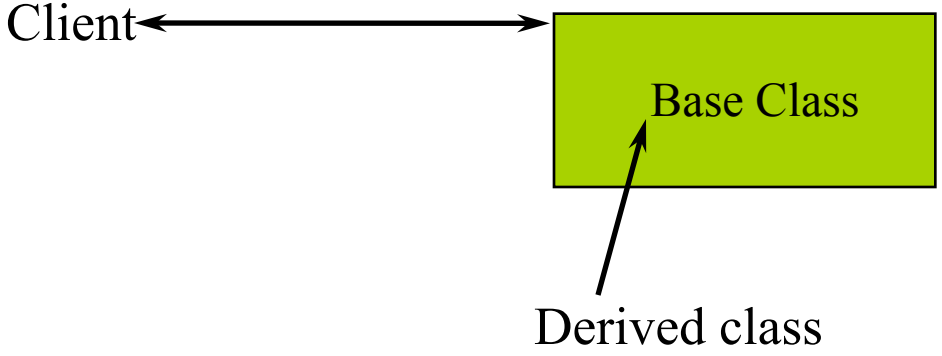
Suppose class B is derived from A. Then

Base class member access specifier

Inheritance Type (B is)	public	protected	private
public A	public in B	protected in B	hidden
private A	private in B	private in B	hidden
protected A	protected in B	protected in B	hidden

When is protected not protected?

- When your derived classes are ill-behaved!
- Protected is public to all derived classes
- For this reason
  - make member *functions* protected
  - keep member *variables* private



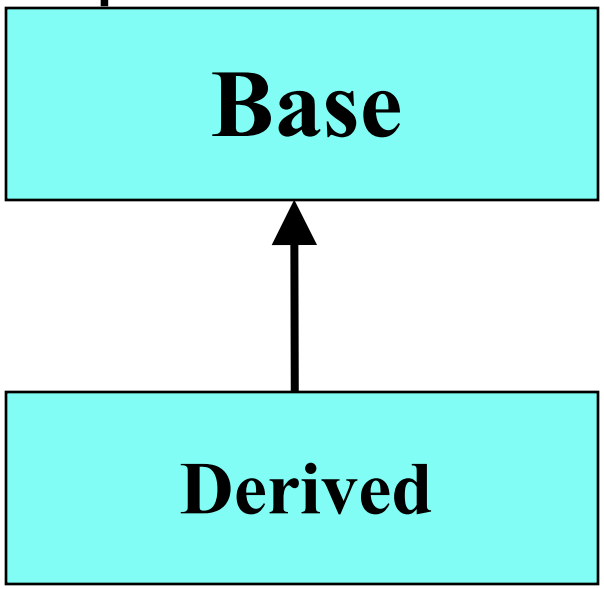
Conversions

- 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	⇒	B
D*	⇒	B*
D&	⇒	B&

Upcasting

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



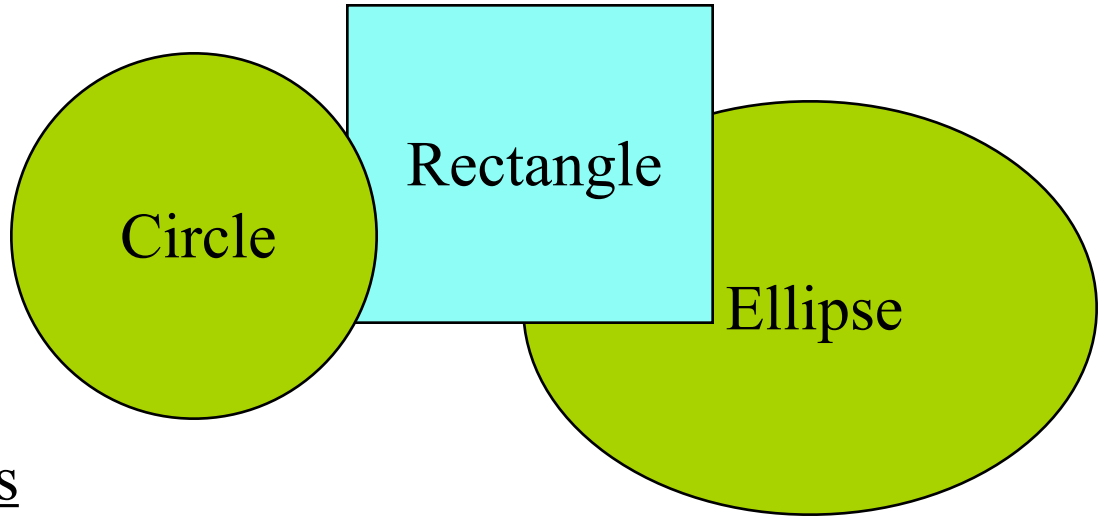
Upcasting examples

```
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



Operations

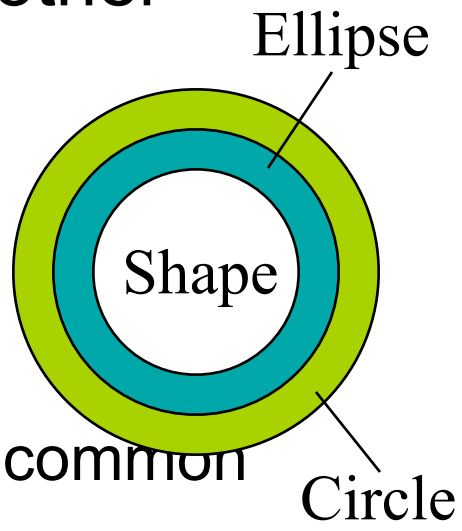
- render
- move
- resize

Data

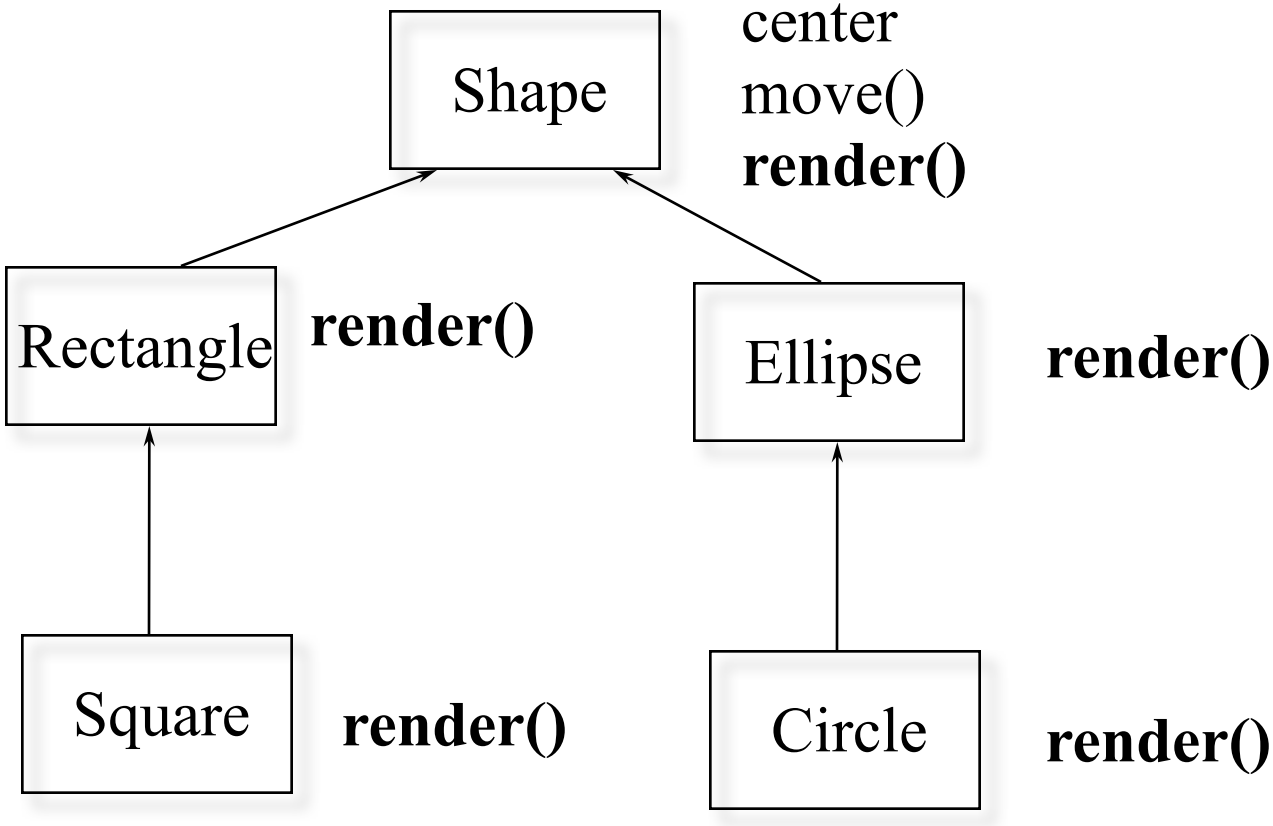
+ center

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!

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;
};
```

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

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()
}
```

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

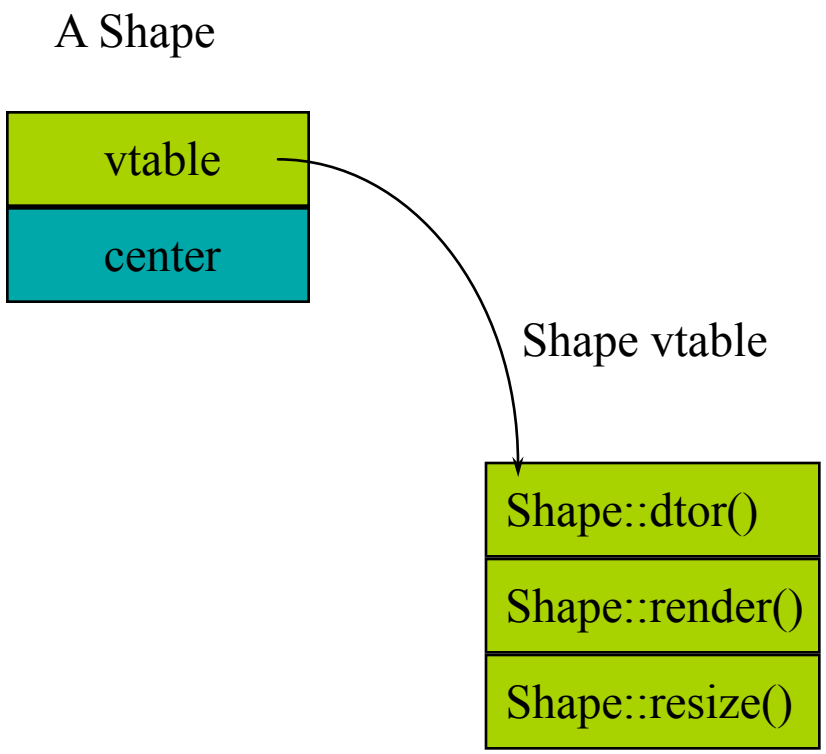
- Unix character device

```
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;
};
```

How virtuals work in C++

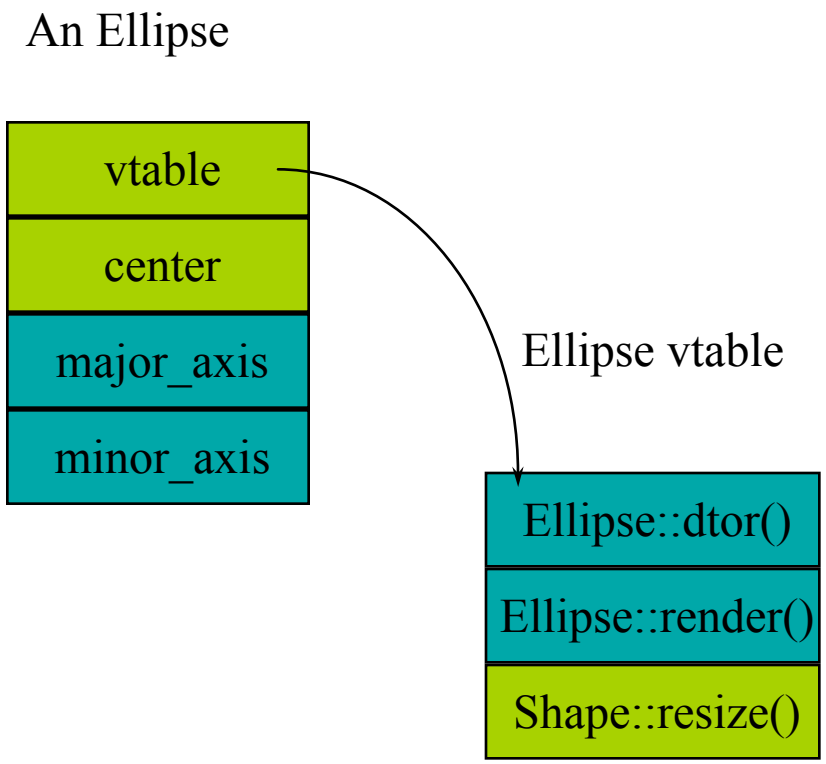
```
class Shape {
public:
   Shape();
   virtual ~Shape();
   virtual void render();
   void move(const
    XYPos&);
   virtual void resize();
protected:
   XYPos center;
};
```



See: Virtual.cpp

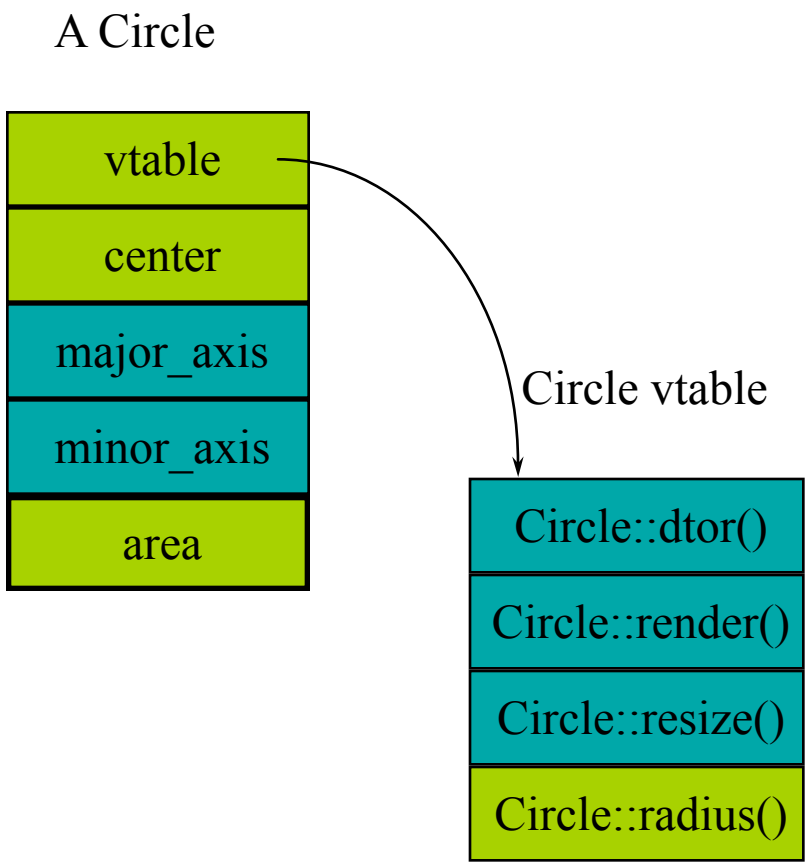
Ellipse

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



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 virtuals

- 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()"; }
};
```