目录

第7章 Polymorphism 多态

7.1 类型转换 Conversions

(1)**public**继承，应该包含替换：即将子类转成父类(子类是父类的超集)

(a)如果B是A的子类，那么能使用A的地方，一定能使用B

(b)如果B是A的子类，那么对A成立的，对B也成立

```
class A{
    int m_a;
public:
    int getA(){m_a = 0; return m_a;}
}
class B : public A{
    int getA(){m_a = 10; return m_a;}
}
void func(A a){
    cout << getA();
}
int main(){
    B b_ins;
    func(b_ins);
    //此时,会将b_ins转换为A类型,因此其输出为0
}
```

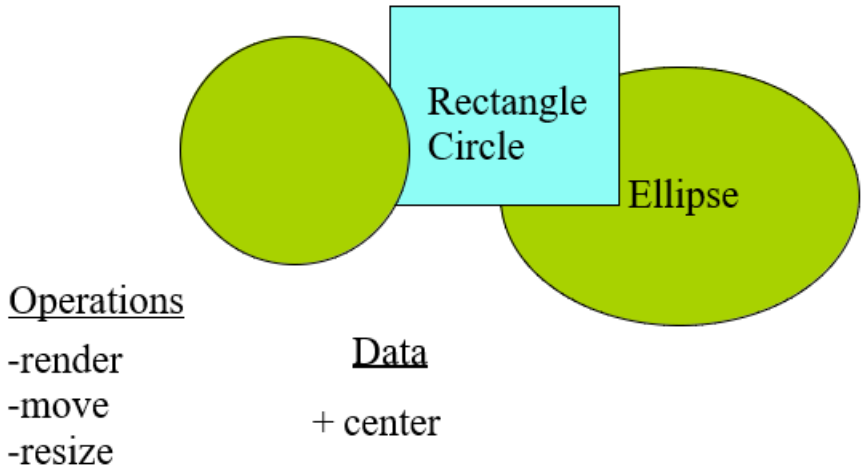
D is derived from B		
D	⇒	B
D*	⇒	B*
D&	⇒	B&

7.1.1 向上转换 Upcast

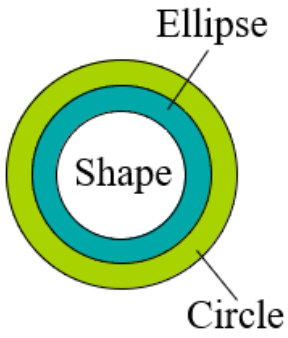
```
Manager pete("Pete", "444-55-6666", "Bakey");
Employee* ep = &pete;   // Upcast
Employee& er = pete;    // Upcast
//会丢失子类额外定义的信息

ep->print( cout );  //调用的是父类的print函数
```

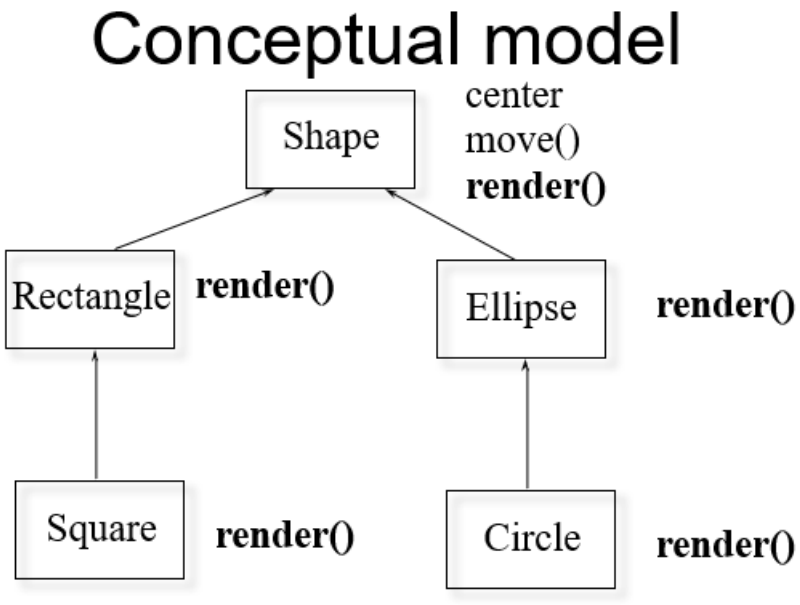
例：drawing

1 具体对象、共有数据、接口



2 继承结构





Note: Deriving Circle from Ellipse is a poor design choice!
Circle需要的变量更少(r)，函数有更快的实现

1. 在父类**Shape**中，定义了接口**render()**
2. 在子类中，通过多态的机制，重定义接口**render()**

3 类的声明

```
class XYPos{ ... }; // x,y point
class Shape {
protected:
    XYPos center;

public:
    Shape();
    virtual ~Shape();
    virtual void render();
    void move(const XYPos&);
    virtual void resize();

};
```

7.2 多态 Polymorphism

向上转换**upcast**：将子类的对象作为父类的一个对象

多态使用的时间：**upcast**

多态的目的：在**upcast**的时候，使用子类的对象

动态绑定**binding**：

1. 绑定**binding**：调用哪个函数
2. 静态绑定**static binding**：调用函数作为代码
3. 动态绑定**dynamic binding**：调用对象的函数

7.2.1 non-virtual function

(1)视为静态绑定，编译时就确定好使用哪个函数

(2)调用较快

7.2.2 virtual function

1. 子类一定要重定义该函数
2. 对象存储了虚函数的信息
3. 编译器会检查、并动态调用正确的函数
4. 编译器优化：如果编译器在编译的时候知道应该调用哪个函数，则会生成一个静态调用

```
class Ellipse : public Shape {
protected:
    float major_axis, minor_axis;

public:
    Ellipse(float maj, float minr);
    virtual void render(); // will define own
};

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

void render(Shape* p) {
    //借助多态机制virtual,函数只需要跟父类打交道,upcast的时候会保证调用子类的对应接口
    p->render();
}

void func() {
    Ellipse ell(10, 20);
    ell.render(); //静态绑定,没有upcast,调用了 Ellipse::render();

    Circle circ(40);
    circ.render(); //静态绑定,没有upcast,调用了Circle::render();

    render(&ell); //动态绑定,出现upcast:Ellips->Shape,会调用Ellipse::render();
    render(&circ); //动态绑定,出现upcast:Circle->Shape,会调用Circle::render();
}
```

7.3 抽象基础类 Abstract base classes

7.3.1 定义

(1) 一个抽象的基础类有纯虚函数**pure virtual functions**

1. 只定义返回值、参数
2. 不需要给函数体

(2) 抽象基础类不能直接实例化≠不能用指针

1. 必须由一个子类继承
2. 子类必须要实现抽象类的所有纯虚函数

7.3.2 例：定义纯虚函数

```
class XYPos{ ... }; // x,y point
class Shape {
protected:
    XYPos center;
public:
    Shape();
    virtual void render() = 0; //告诉编译器，该函数为纯虚函数，子类必须重载该函数
    void move(const XYPos&);
    virtual void resize();
};
```

7.3.3 作用

(1)便于建模 ==> 抽象定义

(2)强制要求正确的行为 ==> 所有子类都必须有该行为

(3)定义接口，而不是定义实现

7.3.4 例：定义抽象类

```
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;
};
```

(1)所有非静态的成员函数都是纯虚函数，除了析构函数

(2)虚拟析构函数，没有函数体

(3)没有非静态的成员变量，可以有**静态成员变量**

7.4 Virtual的实现机制

(1)在内存中，会存储一个指针，指向当前类的虚函数表

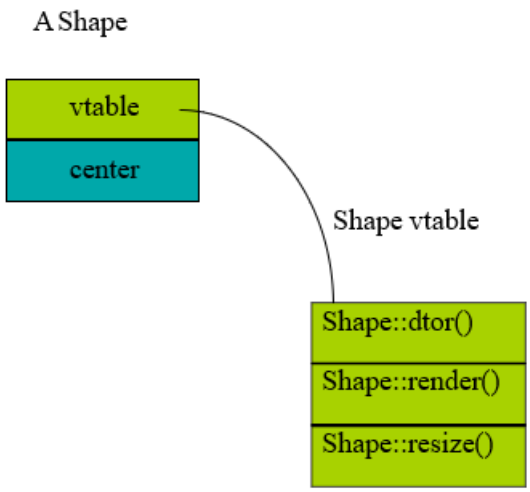
(2)一个class会有一个虚函数表

(3)父类的变量会在子类的变量之前

(4)子类的虚函数表中，不会有父类的虚函数表，因为子类已经将所有虚函数重新定义了

How virtuals work in C++

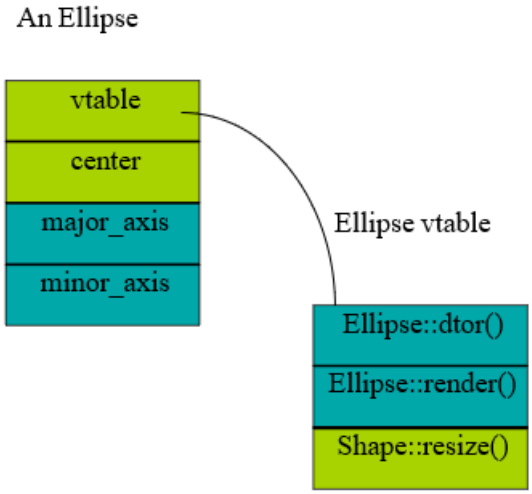
```
class Shape {
public:
   Shape();
   virtual ~Shape();
   virtual void render();
   void move(const
   XYPos&);
   virtual void resize();
protected:
   XYPos center;
};
```



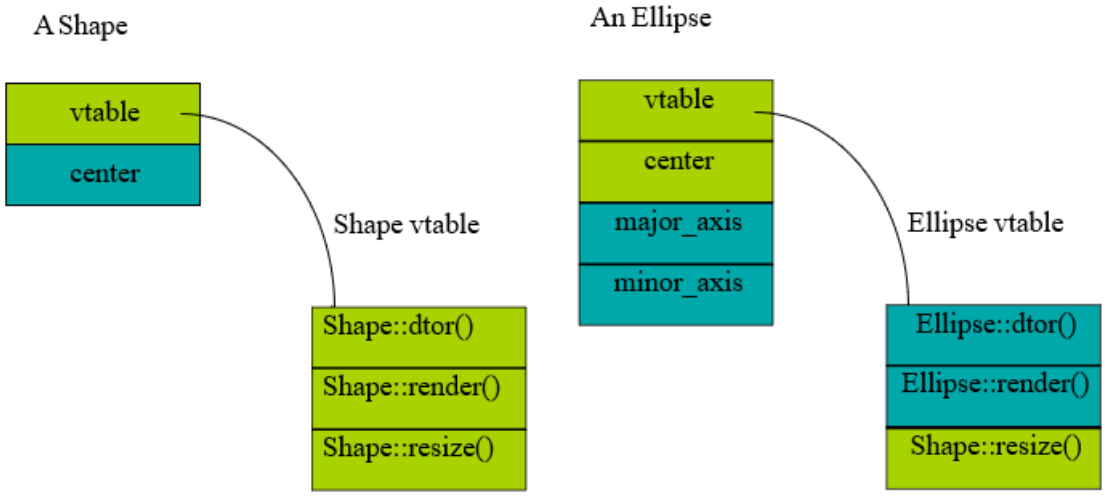
see: virtual.cpp

Ellipse

```
class Ellipse: public Shape{
public:
     Ellipse(float maj, foat
minr);
     virtual void render();

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

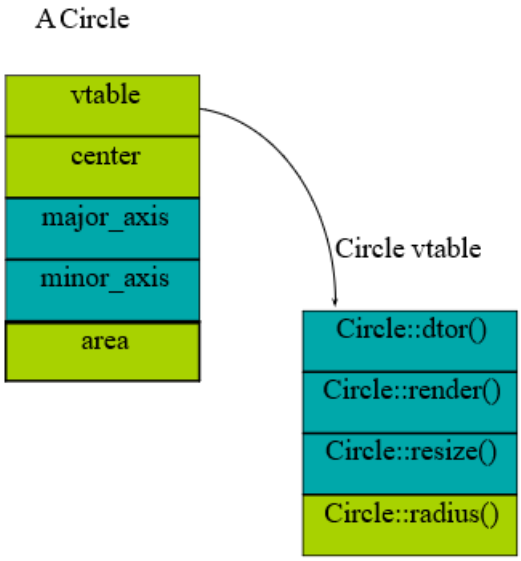


Shape vs Ellipse



Circle

```
class Circle: public Ellipse{

public:
     Circle(float radius);
     virtual void render();
     virtual void resize();
     virtual float radius();

protected:
     float area;

};
```



7.5 赋值与upcast

```
Ellipse elly(20f,40f);
Circle  circ(60f);
elly = circ;
elly.render(); //Ellipse::render()
//赋值的时候,将circ与elly共有的属性赋给elly
//但是elly的虚函数表不会变
//elly.render()时,调用的依旧是父类的render()

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

```
elly = circ;
elly->render(); //Circle::render()
//指针赋值,对应的内存并不会变,但是出现了upcast
//elly->render()时,会找到子类的虚函数表,然后执行子类的render()函数
```

7.6 Relaxation

虚函数的返回类型为指针、引用的时候，子类可以修改返回类型

虚函数的返回类型为class的时候，子类不能修改返回类型

```
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
}
```

7.7 Overloading and virtuals

重载的几个函数，都需要被子类重定义

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

7.8 注意

(1)不要重新定义**non-virtual**的函数

(2)不要重定义继承函数的缺省值

```cpp
class A {
public:
    A() { f();} //A::f()
    virtual void f() { cout << "A::f()"; }
};
class B : public A {
public:
    B() { f();}//B::f()
    void f() { cout << "B::f()"; }
};
// 构造B的时候，会先调用A的构造函数，输出A::f()，然后调用B的构造函数，输出B::f()
```