Object-Oriented Programming

VTables

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Problem for Dynamic Dispatch

```cpp
// Shape files
class Shape {
public:
    virtual void draw();
};

void apply(Shape& shape) {
    // NOTE: virtual method call
    base.draw();
}

void apply(Shape* shape) {
    // NOTE: virtual method call
    pbase->draw();
}

// Circle files
class Circle : public Shape {
public:
    void draw() override;
};

Circle circle;
apply(circle);
apply(&circle);
```

- New methods now exist
- Original `apply()` code **existed and was compiled long before class Circle was created**
- The original `apply()` code in this case **must call `Circle::draw()`**

```
// Shape files
class Shape {
public:
    virtual void draw();
};

void apply(Shape& shape) {
    // NOTE: virtual method call
    base.draw();
}

void apply(Shape* shape) {
    // NOTE: virtual method call
    pbase->draw();
}

// Circle files
class Circle : public Shape {
public:
    void draw() override;
};

Circle circle;
apply(circle);
apply(&circle);
```

How?

- At compile time, the *virtual table* or *vtable* is created to store the information required for the dynamic dispatch of virtual methods for a class
- For every object of a class that has a *vtable*, the object has a pointer to the class vtable
- Method calls go through the vtable for the class of an object

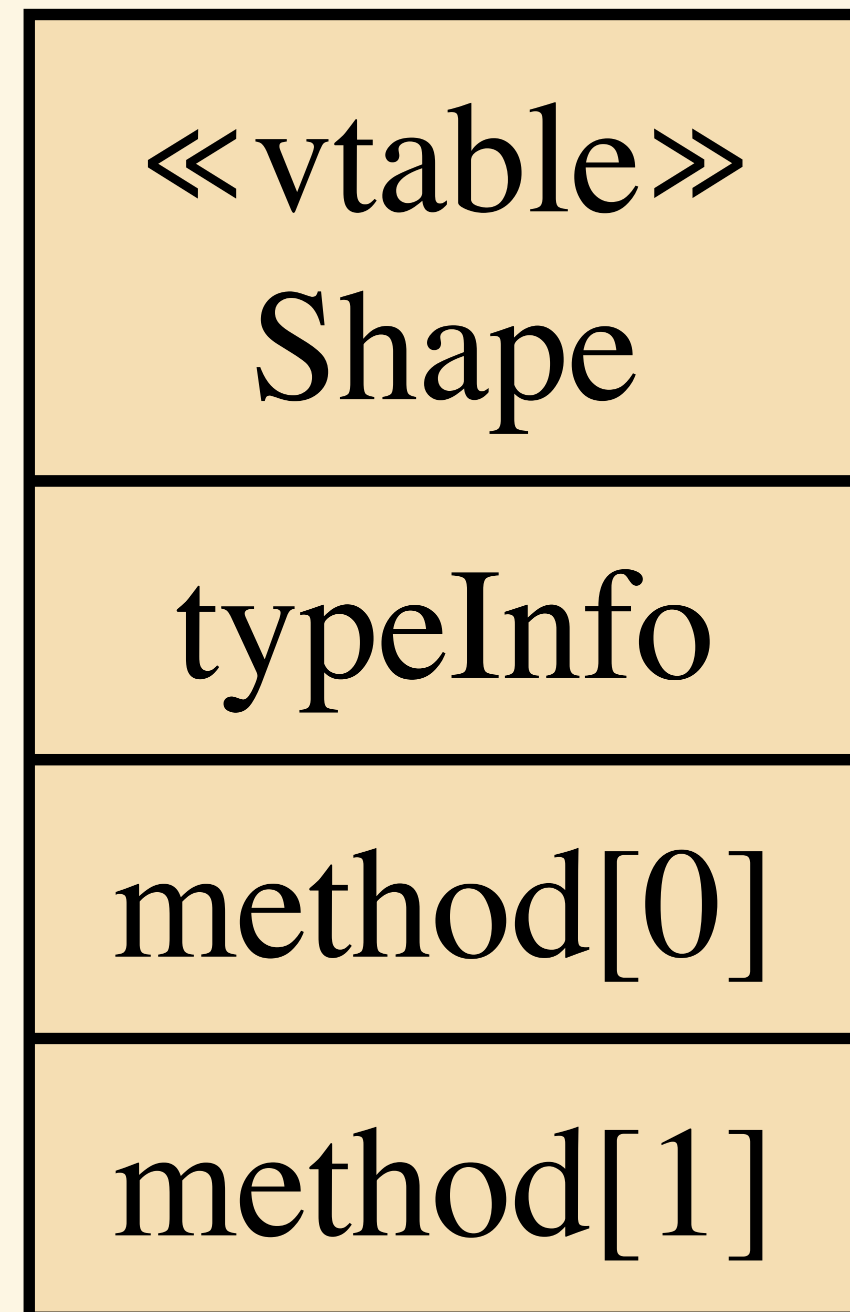
Fundamental Theorem of Software Engineering (FTSE)

We can solve any problem by introducing an extra level of indirection

All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection

- Original quote David Wheeler
- Addendum possibly by Andrew Koenig
- Use of *namespace alias*
- Use of *typedef*
- Use of *references*
- Use of *pointers*

vtable for class Shape with Added Methods




```
class Shape {
public:
    void f();
    virtual void draw();
    virtual void move();
    static void s();
};
```

vtable for class Shape

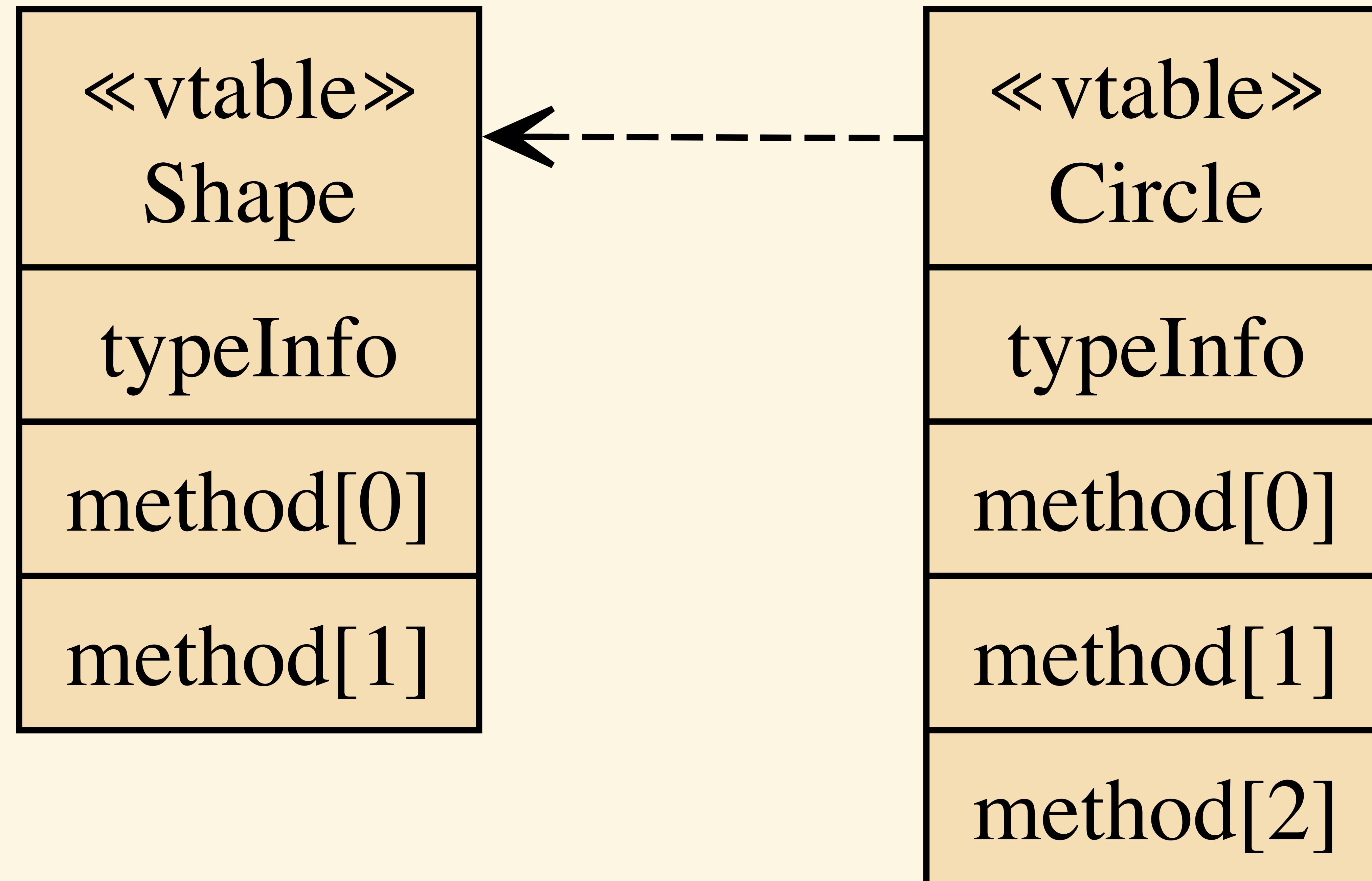
«vtable» Shape
typeInfo
method[0]
method[1]

- A vtable has:
- *typeInfo*
- Array of pointers to virtual methods
- Virtual method calls for dynamic dispatch are stored in the program as an index into an array

  `method[0]` for the first virtual method

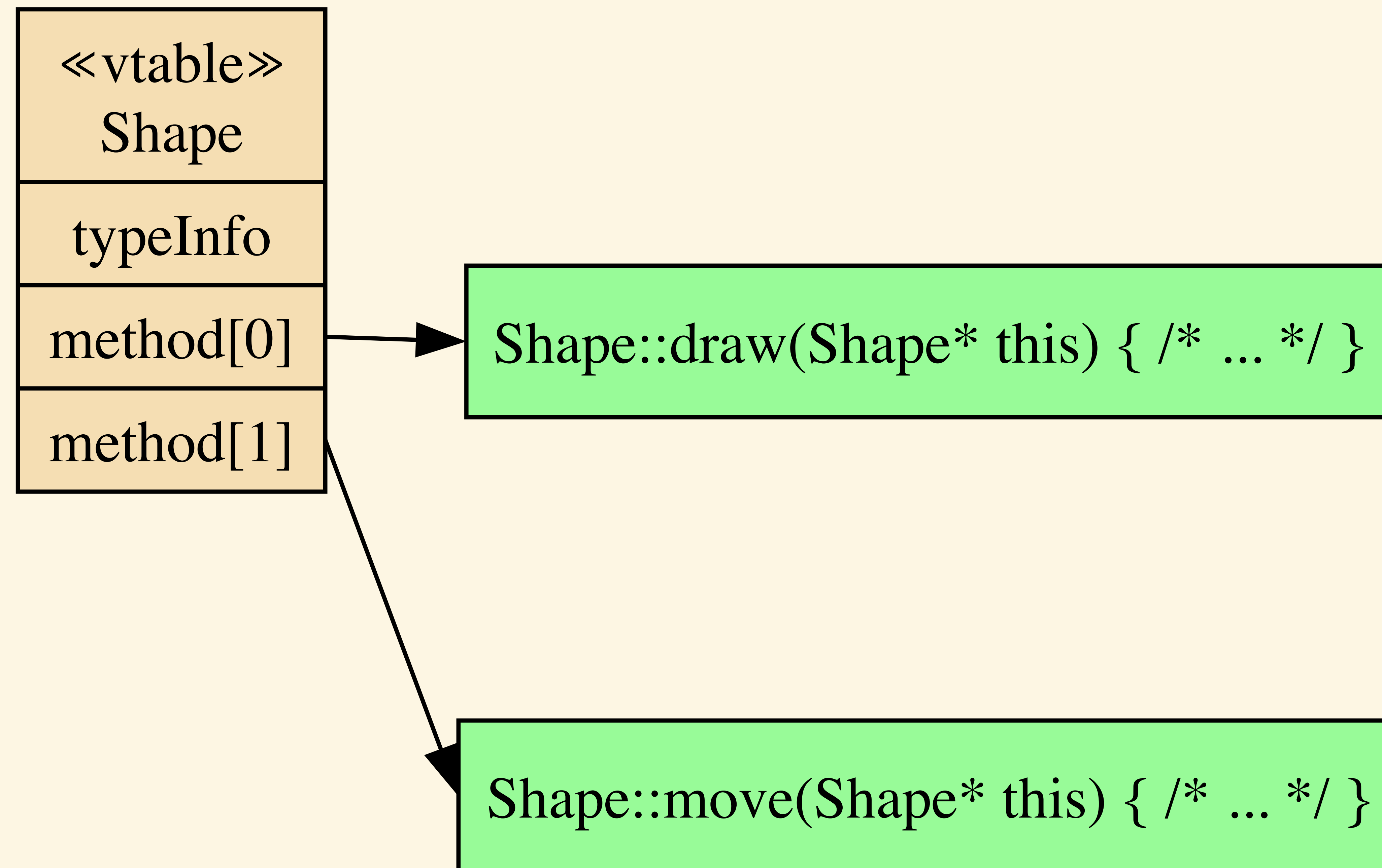
  `method[1]` for the second virtual method

vtable for class Circle



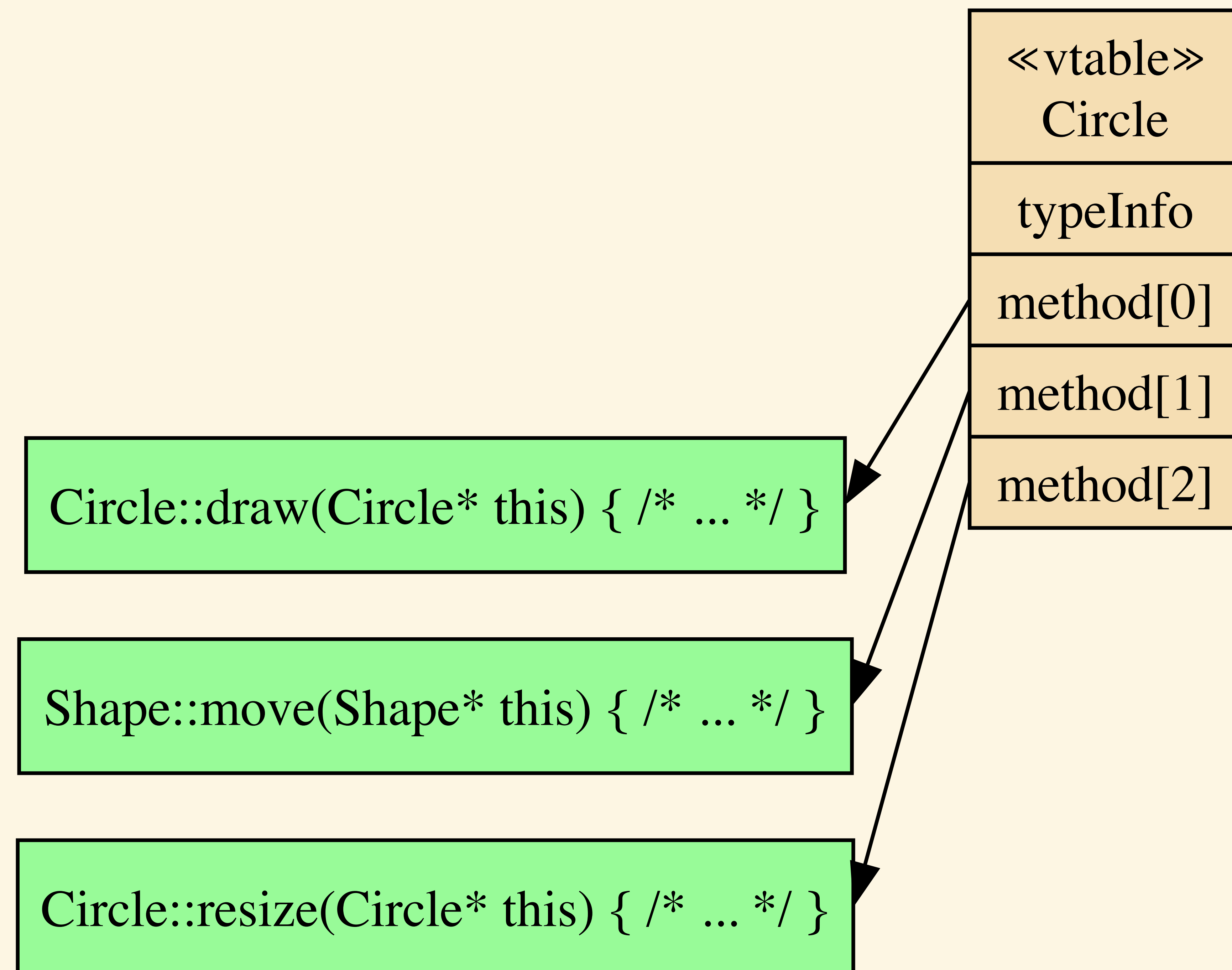

```
class Circle : public Shape {
public:
    void draw() override;
    virtual void resize();
};
```

Base class Shape vtable & functions



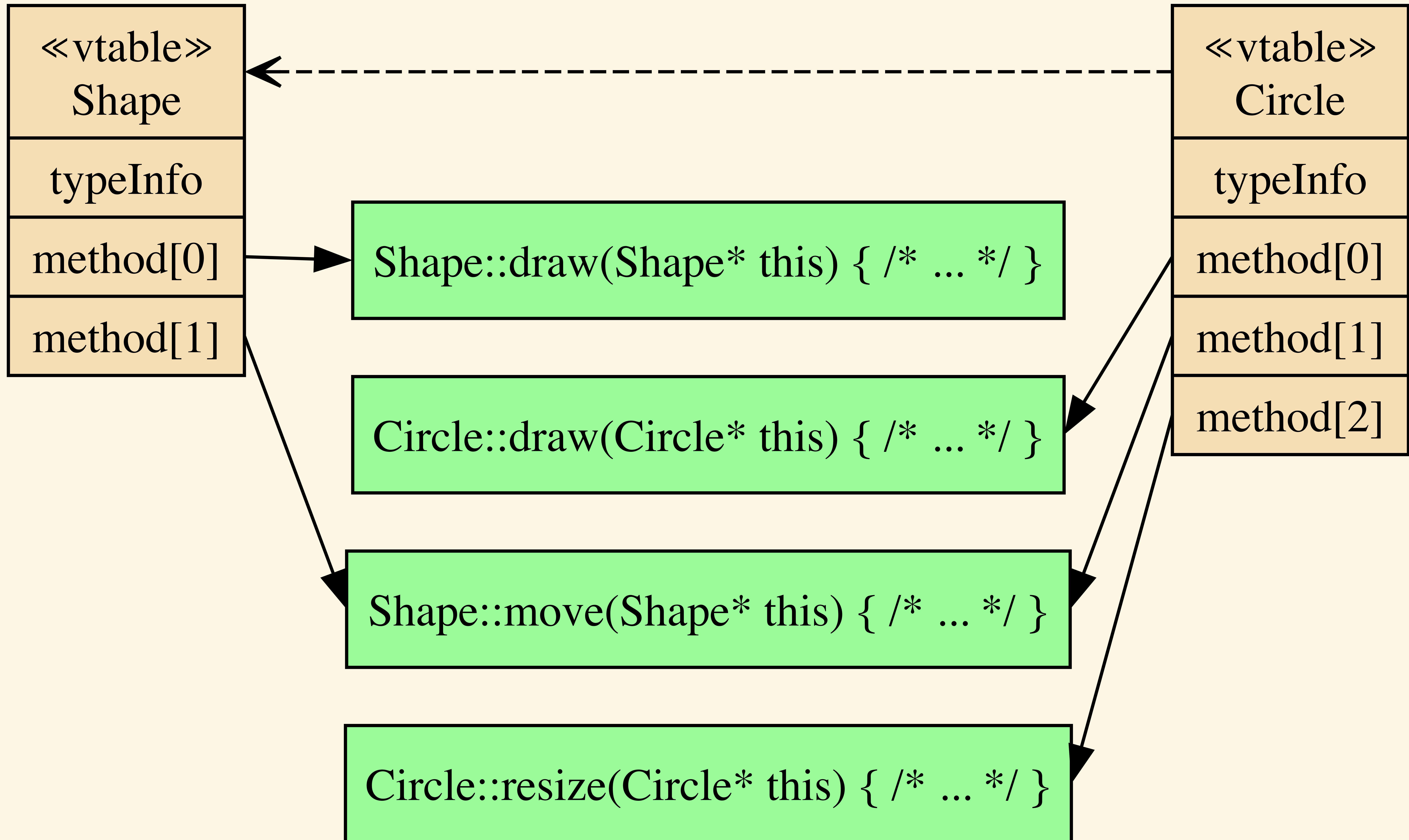

```
class Shape {
public:
    virtual void draw();
    virtual void move();
};
```

Derived class Circle vtable & functions

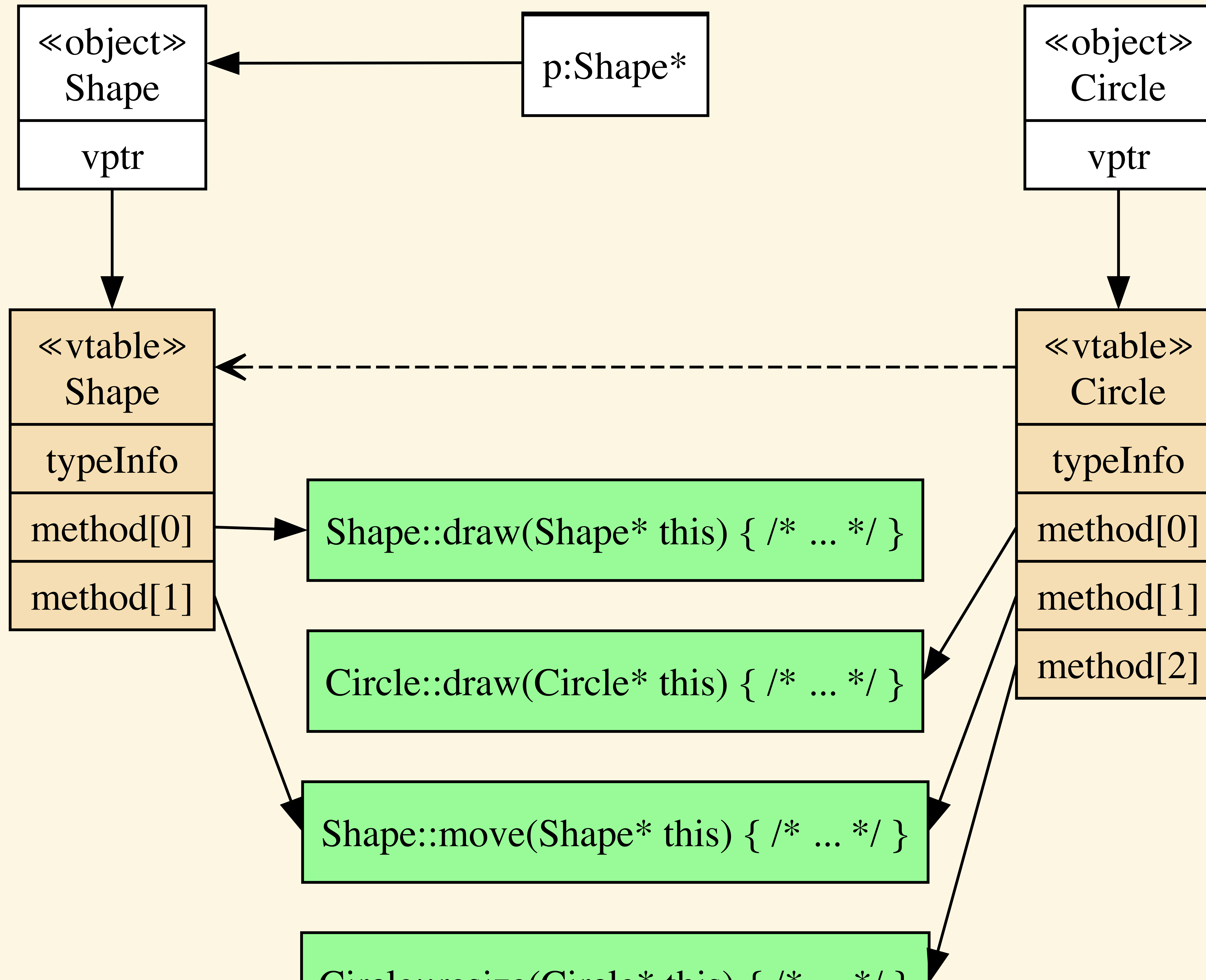



```
class Circle : public Shape {
public:
    void draw() override;
    virtual void resize();
};
```

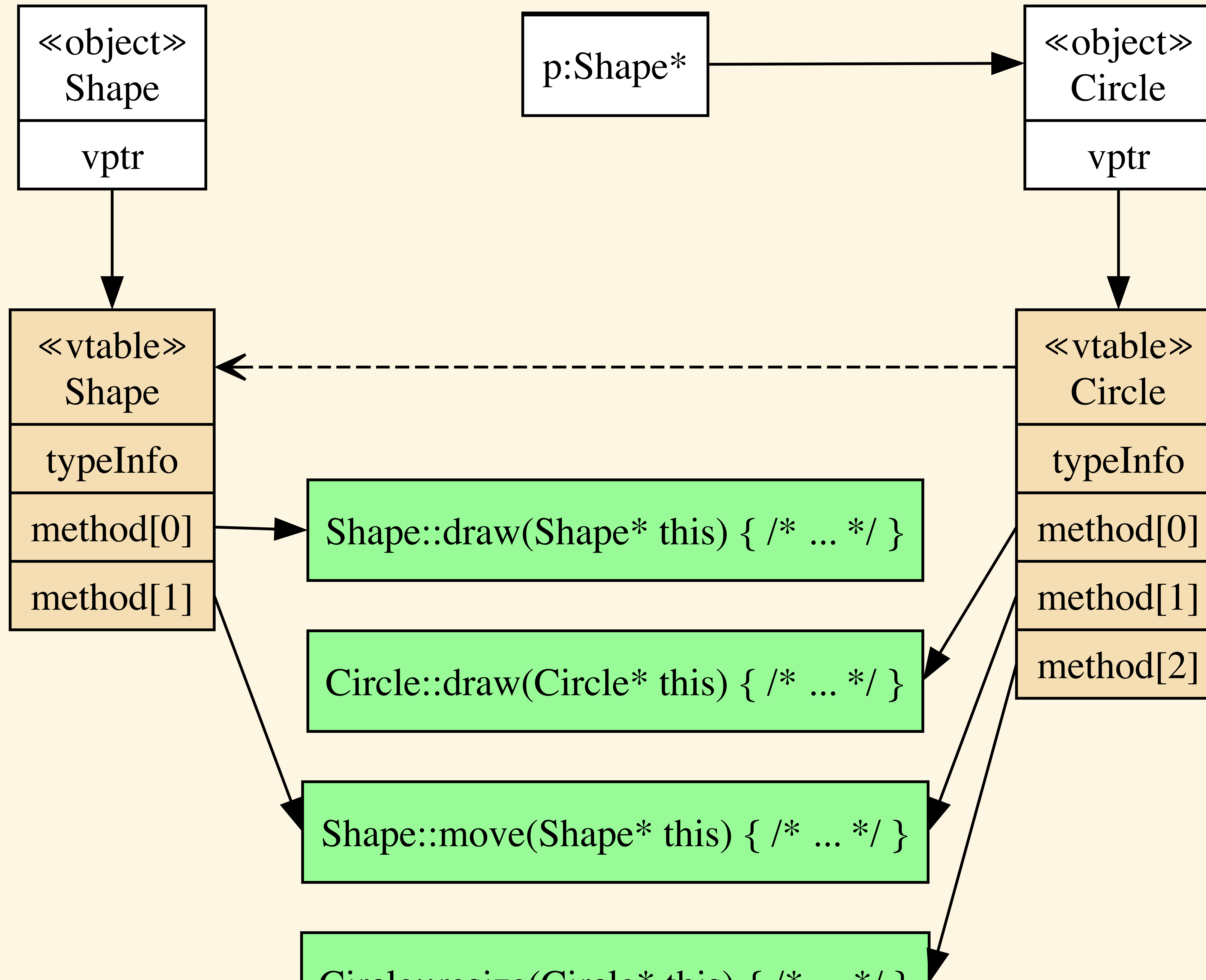
Combined vtables & functions



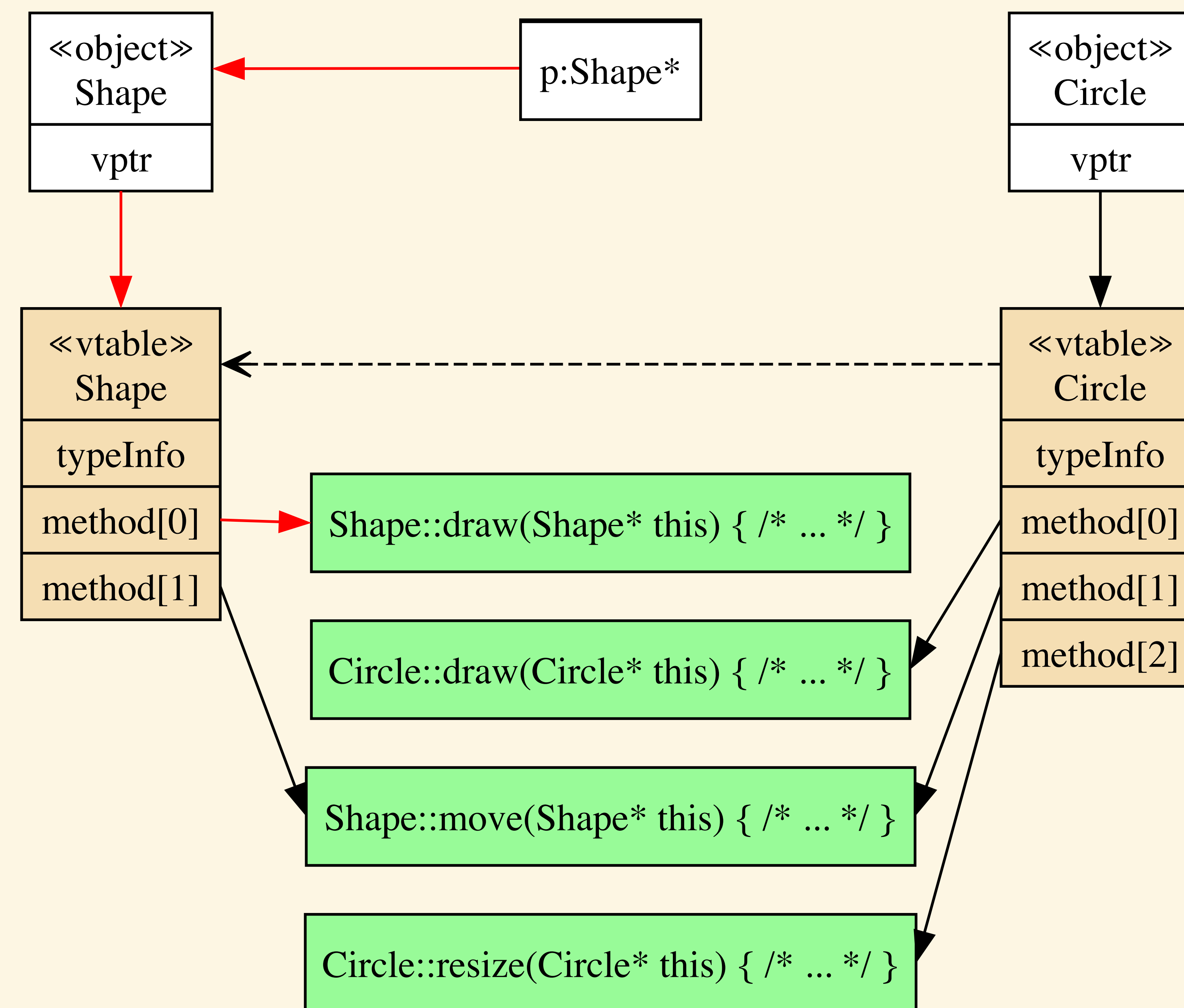
Shape* to Shape Object



Shape* to Circle Object

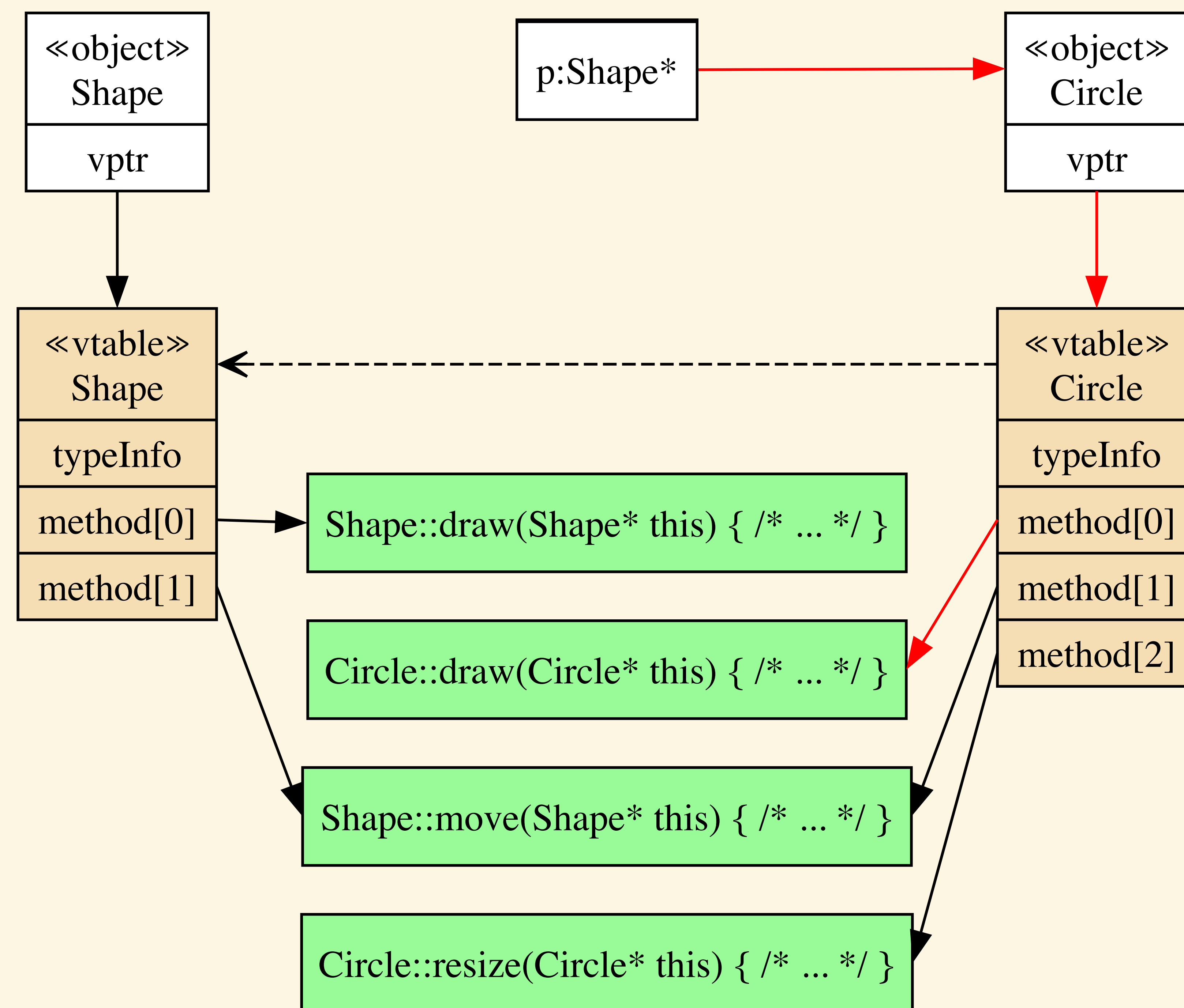


`Shape* p = new Shape(); p->draw();`



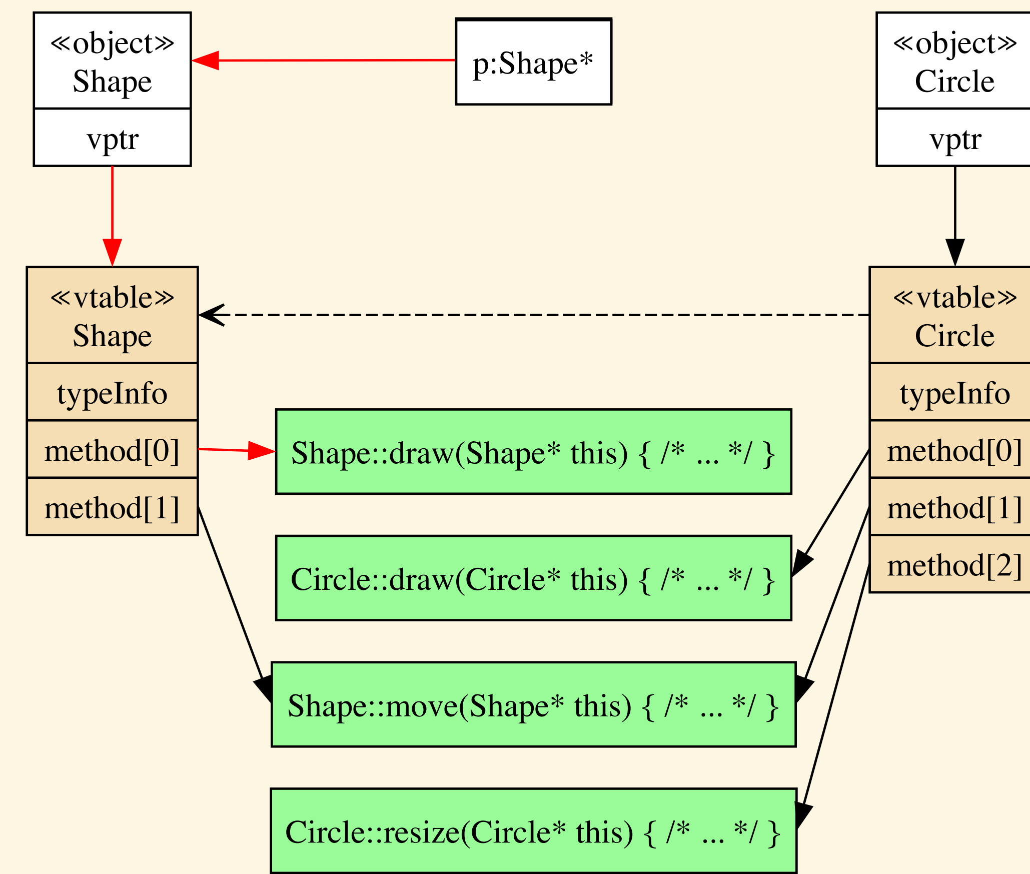
- `Shape* p` points to the new Shape object
- The Shape object points to the vtable for Shape
- The method `draw()` is the first entry in the vtable
- `p->draw()` conceptually becomes `p->vptr->method[0]()` as the first entry in the vtable points to the function for `Shape::draw()`

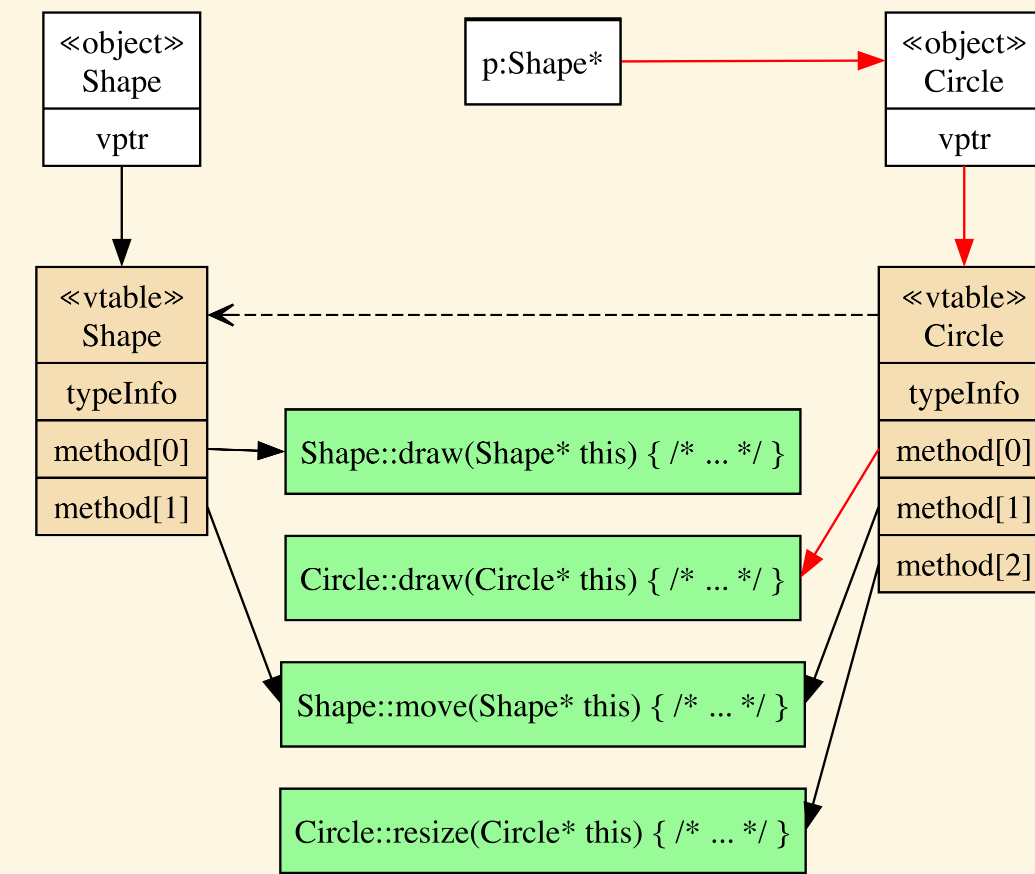
`Shape* p = new Circle(); p->draw();`



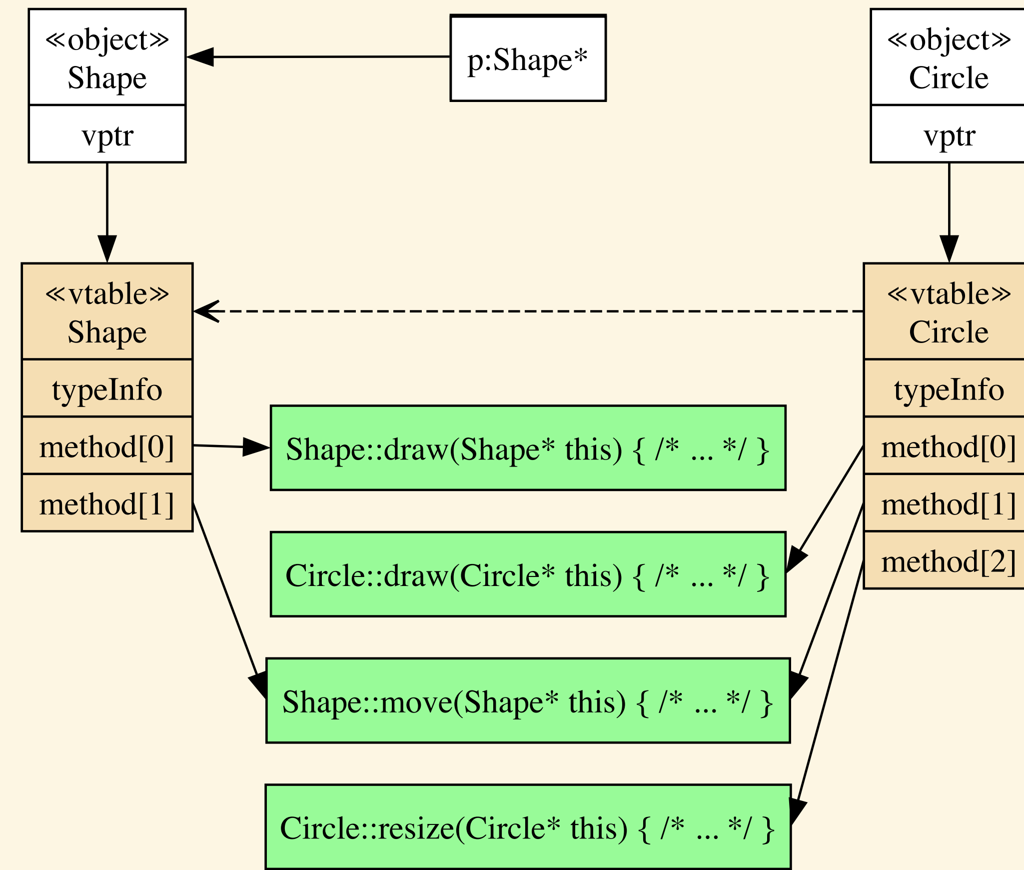
- `Shape* p` points to the new Circle object
- The Circle object points to the vtable for Circle
- The method `draw()` is the first entry in the vtable
- `p->draw()` conceptually becomes `p->vptr->method[0]()` as the first entry in the vtable points to the function for `Circle::draw()`
- The first entry in the vtable points to the function for `Circle::draw()`

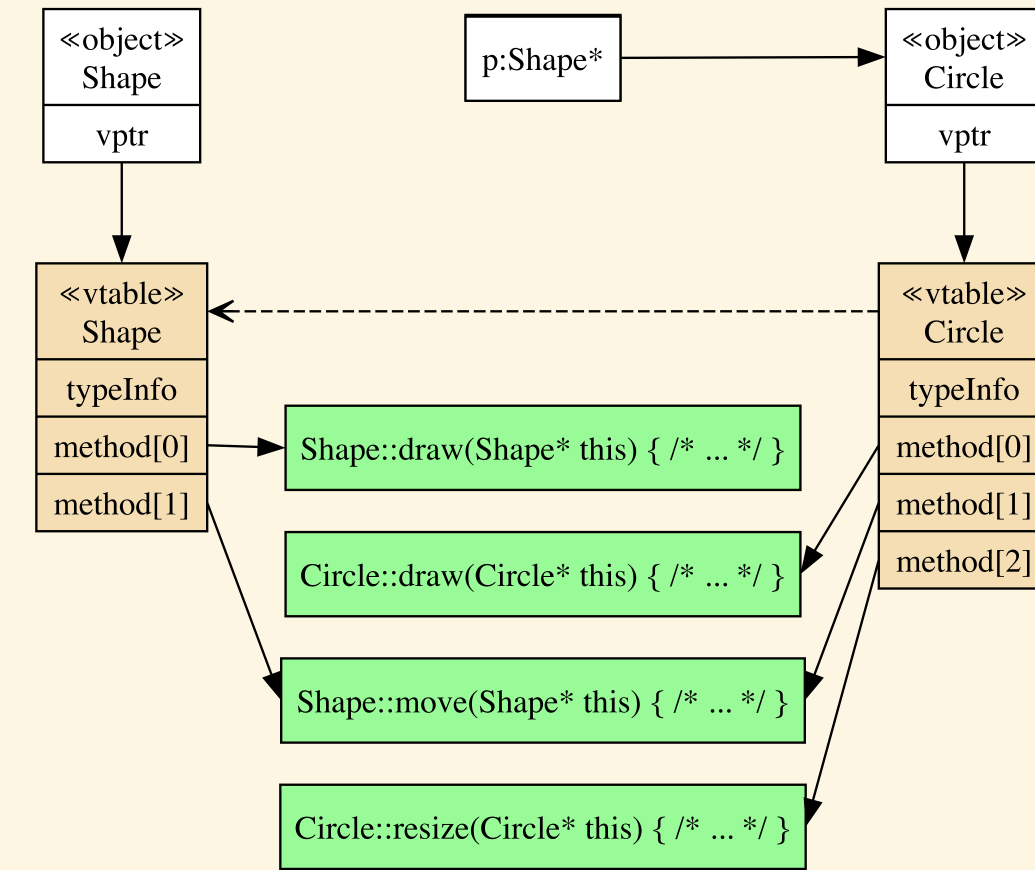
p->draw()



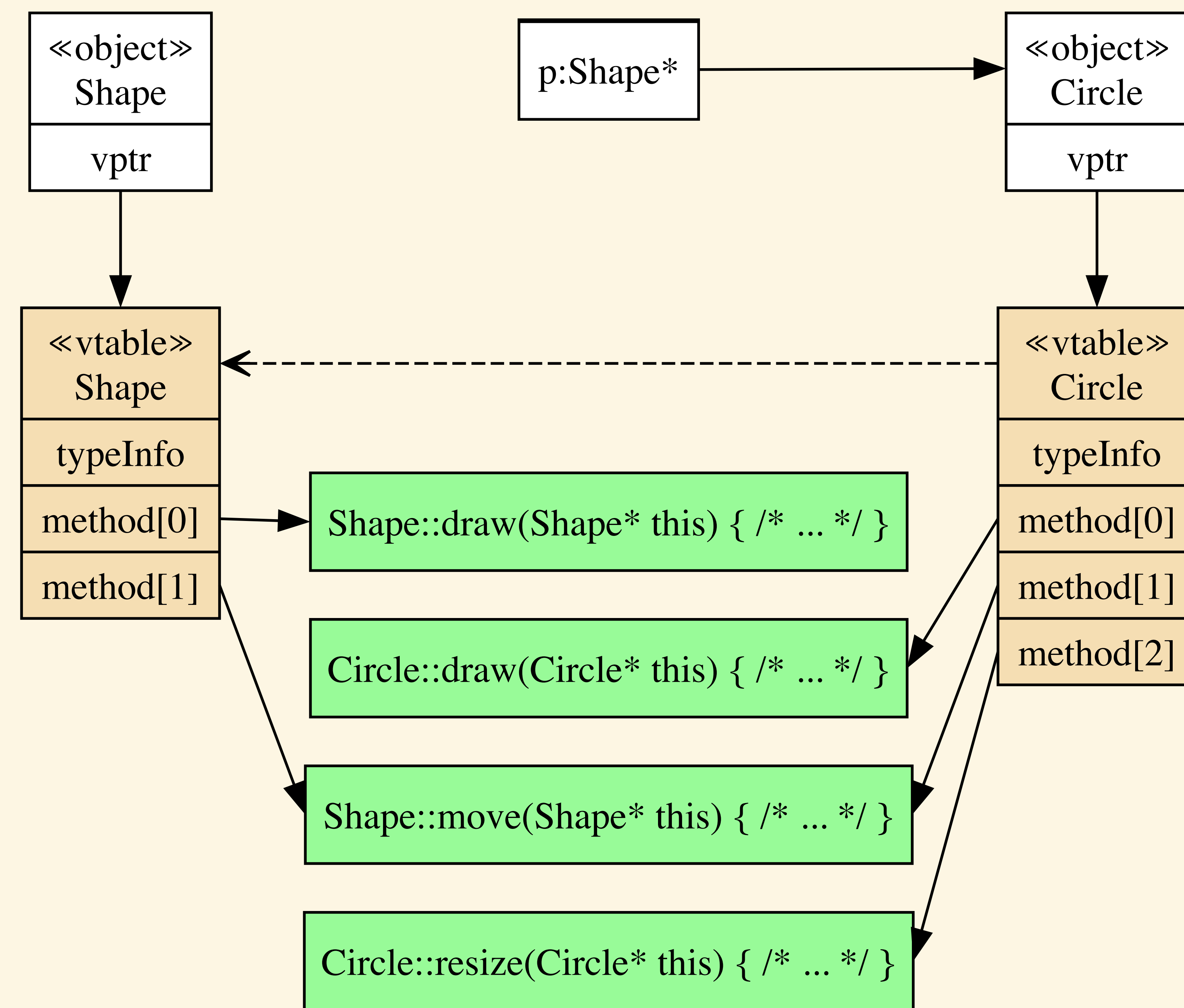


Comparison



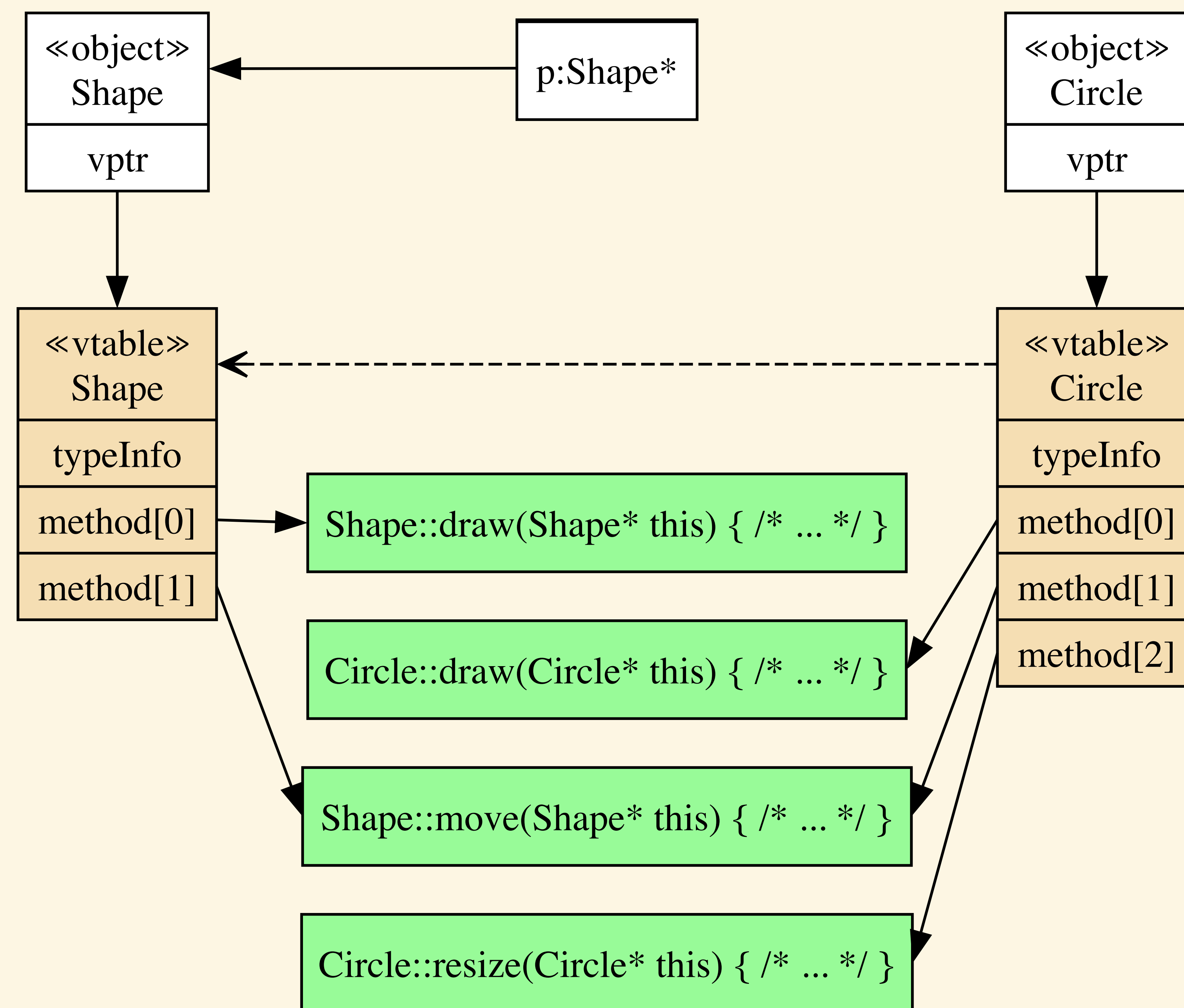


Circle Notes



- There is a dependency for the derived class `Circle` on the base class `Shape`. Changes to the vtable of the class `Shape` (e.g., add, remove, or reorder virtual methods) change the vtable of the class `Circle`.

Shape Notes



- Code compiled with the `Shape` class (e.g., `apply()`) can only call class `Shape` methods, i.e., it cannot call the method `Circle::resize()`.

Conclusion

- *vtable* mechanism supports polymorphism and virtual method calls, even when the derived class methods are not defined yet
- Overhead of a pointer indirection
- Overhead of preventing inlining
- Necessary for *polymorphism*
- But it is worth it in terms of code flexibility