Item 31. Covariant Return Types

Generally, an overriding function must have the same return type as the function it overrides:

```cpp
class Shape {
    public:
        //...
        virtual double area() const = 0;
        //...
};
class Circle : public Shape {
    public:
        float area() const; // error! different return type
        //...
};
```

However, this rule is relaxed for what are known as "covariant return types." If \( B \) is a class type, and a base class virtual function returns \( B * \), then an overriding derived class function may return \( D * \), where \( D \) is publicly derived from \( B \). (That is, \( D \) is-a \( B \).) If a base class virtual function returns \( B & \), then an overriding derived class function may return \( D & \). Consider the following clone operation on a shape hierarchy (see Virtual Constructors and Prototype [29, 99]):

```cpp
class Shape {
    public:
        //...
        virtual Shape *clone() const = 0; // Prototype
        //...
};
class Circle : public Shape {
    public:
        Circle *clone() const;
        //...
};
```

The overriding derived class function is declared to return a \( \text{Circle} * \) rather than a \( \text{Shape} * \). This is legal because \( \text{Circle} \) is-a \( \text{Shape} \). Note that the \( \text{Circle} * \) return value from \( \text{Circle}::\text{clone} \) is automatically converted to \( \text{Shape} * \) if the \( \text{Circle} \) is being manipulated as a \( \text{Shape} \) (see Meaning of Pointer Comparison [28, 97]):

```cpp
Shape *s1 = getACircleOrOtherShape();
Shape *s2 = s1->clone();
```

The advantage of using covariant return types comes when manipulating derived types directly, rather than through their base class interfaces:

```cpp
Circle *c1 = getACircle();
Circle *c2 = c1->clone();
```
Without a covariant return, `Circle::clone` would have to match exactly the return type of `Shape::clone` and return a `Shape *`. We'd be forced to cast the return result to `Circle *`.

```cpp
Circle *c1 = getACircle();
Circle *c2 = static_cast<Circle*>(c1->clone());
```

As another example, consider the following Factory Method member of `Shape` that returns a reference to an appropriate shape editor for the concrete shape (see `Factory Method` [30, 103]):

```cpp
class ShapeEditor { ... };
class Shape {
public:
    //...
    virtual const ShapeEditor &
        getEditor() const = 0; // Factory Method
    //...
};
//...
class Circle;
class CircleEditor : public ShapeEditor { ... };
class Circle : public Shape {
public:
    const CircleEditor &getEditor() const;
    //...
};
```

In this case, note that `CircleEditor` had to be completely defined (not simply declared) prior to the declaration of `Circle::getEditor`. The compiler has to know the layout of the `CircleEditor` object so it can perform the appropriate address manipulations to convert a `CircleEditor` reference (or pointer) into a `ShapeEditor` reference (or pointer). See `Meaning of Pointer Comparison` [28, 97].

The advantage of the covariant return is that we can always work at the appropriate level of abstraction. If we're working with `Shapes`, we'll get an abstract `ShapeEditor`; if we're working with a specific type of shape, such as `Circle`, we'll be able to deal directly with `CircleEditors`. The covariant return relieves us from having to use an error-prone cast to resupply type information that we should not have lost in the first place:

```cpp
Shape *s = getACircleOrOtherShape();
const ShapeEditor &sed = s->getEditor();
Circle *c = getACircle();
const CircleEditor &ced = c->getEditor();
```