

## Inheritance

Capturing similarities between types

## Overview

- This lecture and the next one will describe object-oriented programming
- Most of the ideas described here are useful in more than one language
- Specific examples, as usual, are in C++

## Philosophical note

- Abstraction is selective ignorance
- How do you decide what to ignore?
- One way is to note similarities and differences among several things
  - Sometimes, you want to concentrate on the similarities
  - Other times, you want to ignore the similarities and look only at the differences

## Inheritance

- Inheritance is a way of describing a class by saying how it differs from another class
- Example: "Class Y is just like class X except for the following differences..."
  - Class Y is called a *derived class* or *subclass*
  - Class X is called a *base class* or *superclass*

## Why use inheritance?

- The usual reason is when you have two types where one is necessarily an extension of the other
- Sometimes (but not all the time) you are going to want to ignore the differences and look only at the base class (which is what they have in common)

## The classic example

- Consider a system that can manipulate various kinds of shapes
- Sometimes you don't care what particular kind of shape you have (example: move to a different location)
- Sometimes you do care (example: draw the shape on a display)

## Specifying inheritance in C++

```
class Shape {  
public:  
    Point position;  
    // ...  
};  
class Circle: public Shape {  
public:  
    int radius;  
    // ...  
};
```

## What it means

- When we say  

```
class Circle: public Shape { /* ... */ };
```

  
we are saying that
  - A `Circle` is a kind of `Shape`
  - Therefore, in addition to its own members, class `Circle` inherits all the members of class `Shape`, and
  - The fact that a `Circle` is a kind of `Shape` is publicly available

## When to use inheritance

- When you want to be able to say “Every `Y` is really a kind of `X` with some extra properties”
- When you really want `Y` to be able to do everything `X` can do
- This state of affairs leads to...

## The Liskov substitution principle

- If a class `Y` is “just like” a class `X` except for extensions, then it should be possible to use a `Y` object anywhere you can use an `X` object
- You should design your classes to preserve that property unless you have a strong reason to do otherwise

## Examples

- An aircraft is a kind of vehicle
- An airplane is a kind of aircraft
- So is a helicopter
- A square is a kind of shape
- So is a triangle
- So is a generalized polygon
- Is a square a kind of polygon?

## Is a square a (kind of) polygon?

- The answer depends on whether we can follow the Liskov principle:
  - Suppose we have a program that uses polygon objects
  - The Liskov principle says that we should be able to rewrite the program using square objects instead
  - Can we do that?

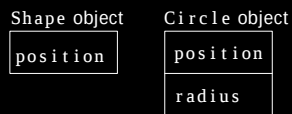
## Squares and polygons

- Whether a (class that represents a) square is a kind of a (class that represents a) polygon depends on what properties of polygons we're capturing
  - If a polygon object contains a list of sides, the answer is probably no
  - If a polygon object contains just a position, and implies that there are no curves, the answer might be yes

## Other examples

- A circle is not a kind of ellipse, nor is an ellipse a kind of circle
- A square is not a kind of rectangle, nor is a rectangle a kind of square
- But an immutable square might be a kind of rectangle, and an immutable circle a kind of ellipse

## Implementation



## Access to base class

- Derived class members can access protected and public members of base-class object
- Pointer (or reference) to derived can be converted to pointer (or reference) to public base (or, within member function body, to protected base)

## Inheritance of members

- Every public member of the base class is a member of the derived class

```
Circle c;  
// ...  
Point p = c.position;
```

## Classes are scopes

- A member of a derived class hides all base-class members with that name

```
class X {  
    public: void f(int);  
};  
class Y: public X {  
    public: void f(char); // hides X::f  
};  
Y y;  
y.f(123456); // calls Y::f(char)
```

## Conversion examples

```
Shape s;  
Circle c;  
Shape* sp = &c;    // OK  
Circle* cp = &s;    // Ill-formed  
Shape& s1 = c;      // OK  
Circle& c1 = s;     // Ill-formed  
s = c;              // OK  
c = s;              // Ill-formed
```

## Why allow `s = c`?

- Class Shape implicitly has a `Shape::Shape(const Shape&)` copy constructor and an analogous assignment operator
- We can bind a `const Shape&` parameter to (the Shape part of) a Circle object
- Only the Shape part is actually copied

## A tiny vector class

```
class Vector {  
public:  
    Vector(int n): data(new int[n]) { }  
    ~Vector() { delete[] data; }  
    int& operator[](int n) {  
        return data[n];  
    }  
private:  
    int* data;  
};
```

## A vector class with explicit bounds

```
class BVec: public Vector {  
public:  
    BVec(int begin, int end):  
        b(begin), Vector(end-begin) { }  
    int& operator[](int n) {  
        return Vector::operator[](n-b);  
    }  
private:  
    int b;  
};
```

## Treating a BVec as a Vector

```
// Sum the first n Vector elements  
int sum(Vector& v, int n)  
{  
    int r = 0;  
    for (int i = 0; i < n; ++i)  
        r += v[i];  
    return r;  
}  
  
BVec b(10, 40);  
int s = sum(b, 20); // [10, 30)
```

## Why does this example work?

- A BVec is a kind of Vector
- Calling `sum(b, 20)` binds `v` to the Vector part of `b`
- When `sum` is running, it doesn't care whether it's working on a Vector, a BVec, or an object of some other class derived from Vector.

### This example is ...

- Unusual: Usually, derived-class operations will not hide base-class operations
- Incomplete: The classes should have copy constructors and assignment operators
- Slightly naughty: It does not follow the Liskov substitution principle

### Where is the violation?

- Remember: A derived class object should be able to substitute for a base class object without changing the behavior of the program
  - A `Circle` should do everything a plain `Shape` can do (but not vice versa)
  - A `Bvec` should do everything a plain `Vector` should do (but not vice versa)
- But we can't create, say, `Bvec(10)`

### Does `operator[]` violate the principle?

- The definitions are definitely different in the base and derived classes
- However, they do the same thing when the lower bound is zero
- A `Vector` has a lower bound of zero
- So there is no problem here
- Note: A base class does not have to substitute for a derived class

### Cleaning up `Bvec`

- The `operator[]` member doesn't violate the Liskov principle
- Therefore, all we really have to do is give `Bvec` a second constructor, with no arguments

### Examples of inheritance

- In chess, a capture is a kind of move
- A `while` statement is a kind of statement
- A manager is a kind of employee
- A directory is a kind of file (though we may want to think about whether this notion follows the Liskov principle)

### Examples where inheritance is inappropriate

- An automobile is not a kind of engine
- An integer array might be a kind of array, but it is *not* a kind of integer

## Squares and polygons

- A square might seem at first to be a kind of polygon, but
  - a polygon can have any number of sides
  - a square is restricted to having four sides
- A polygon might seem to be able to do everything a square can do, but
  - a square has one number (the length of a side) that makes no sense for a polygon
  - a polygon is not substitutable for a square

## Other non-inheritance situations

- Squares and rectangles
- Circles and ellipses
- Strings and file names
  - Not every valid string is a valid file name
  - Therefore, file names cannot be substituted for strings
  - But file names support operations that strings do not

## Review

- Inheritance lets us use a base class to describe properties that are common to several classes
- We can convert a pointer (reference) to a class object into a pointer (reference) to a sub-object whose type is a public base class of the object's class

## What's next

- Suppose we have a pointer to a base class object:  

```
Shape* sp = /* some expression */;
```
- How do we know whether that pointer actually points to a Shape or, say, to a Circle? Why might we care?