Polymorphism
2 Different Uses of Inheritance

- Implementation Inheritance
  - Subclass inherits variables and methods from superclass
  - Code reuse
- Interface Inheritance (a.k.a. "subtyping")
  - Establish an IS-A relationship between subclass and superclass
  - Lets you write code in terms of the superclass that can operate on instances of the subclasses
    - Polymorphism (many forms)
Polymorphism Example

void RedrawScreen(Shape * shapes[], int count) {
    for (int x=0; x < count; ++x) {
        shapes[x]->Draw();
    }
}

void main() {
    Shape * shapes[3];
    shapes[0] = new Line(RED, 0, 0, 10, 10);
    shapes[1] = new Circle(BLACK, 25, 25, 10);
    shapes[2] = new Rectangle(BLUE, 10, 10, 50, 30);
    RedrawScreen(shapes, 3);
}

Does this call Shape::Draw or the Draw method on the object's actual class?
It depends on how you write the Shape class!
Static vs. Dynamic Inheritance

- Let Super be the superclass and Sub be the subclass
  - Sub * sub = new Sub();
  - Super * super = sub;

- Static Inheritance
  - sub->Method(); calls Sub::Method
  - super->Method(); calls Super::Method
  - The method that is called is determined at compile-time based on the type of the pointer variable

- Dynamic Inheritance
  - sub->Method(); calls Sub::Method
  - super->Method(); also calls Sub::Method
  - The method that is called is determined at run-time based on the actual type of the object pointed to by the variable
Static vs. Dynamic Inheritance

**Static Inheritance**

class Super {
    public:
        void Method() {
            ...
        }
};

class Sub : public Super {
    public:
        void Method() {
            ...
        }
};

`super->Method();` calls `Super::Method`

**Dynamic Inheritance**

class Super {
    public:
        virtual void Method() {
            ...
        }
};

class Sub : public Super {
    public:
        virtual void Method() {
            ...
        }
};

`super->Method();` calls `Sub::Method`
Dynamic Inheritance Example

class Shape {
    protected:
        Color color;

    public:
        Shape(Color c) {
            color = c;
        }
        virtual void Draw() {
            return;
        }
        Color GetColor() {
            return color;
        }
    }
};

class Line : public Shape {
    protected:
        int x1, y1, x2, y2;

    public:
        Line(Color c,
            int _x1, int _y1,
            int _x2, int _y2) :
            Shape(c)
            {
                x1 = _x1; y1 = _y1;
                x2 = _x2; y2 = _y2;
        }
        virtual void Draw() {
            // CODE TO DRAW A LINE
            // GOES HERE
        }
};
Dynamic Inheritance Example

Shape * obj = new Line(RED, 0, 0, 10, 10);
assert(obj->GetColor() == RED);
obj->Draw();
delete obj;

This calls Line::Draw
This calls Shape::GetColor

What about this? Which destructor gets called here, ~Shape or ~Line?
It calls ~Shape because the destructors on Shape and Line are not virtual
Is this a problem?
Yes. If Line's destructor deallocates resources (e.g., memory), this will result in a resource leak
Virtual Destructors

class Shape {
protected:
    Color color;

public:
    Shape(Color c) {
        color = c;
    }
    virtual ~Shape() { return; }
    Color GetColor() { return color; }
    virtual void Draw() { return; }
};

class Line : public Shape {
protected:
    int x1, y1, x2, y2;

public:
    Line(Color c, int _x1, int _y1, int _x2, int _y2) :
        Shape(c) {
        x1 = _x1; y1 = _y1;
        x2 = _x2; y2 = _y2;
    }
    virtual ~Line() { return; }
    virtual void Draw() {
        // CODE TO DRAW A LINE
        // GOES HERE
    }
};
Virtual Destructors

Shape * obj = new Line(RED, 0, 0, 10, 10);
assert(obj->GetColor() == RED);
obj->Draw();
delete obj;

Now this will call ~Line instead of ~Shape
Pure Virtual Methods

```cpp
class Shape {
protected:
    Color color;

public:
    Shape(Color c) {
        color = c;
    }
    virtual ~Shape() {
        return;
    }
    virtual void Draw() = 0;
};
```

Shape can't really implement a useful Draw method, so we just make it `pure virtual`

The superclass does not provide a default implementation for a pure virtual method

Trying to call the superclass' implementation of a pure virtual method will crash the program
`Shape::Draw(); // disaster`

You can't make a destructor pure virtual
Why not?
Subclass destructors will always call the superclass destructor, so the superclass needs to implement its destructor
Pure Virtual Methods

- A class that has one or more pure virtual methods is called an **abstract class**
- You can't create instances of an abstract class because it doesn't implement all of its methods
- Abstract classes can only be used as superclasses
- Example: Shape
Interfaces

- A class that only contains pure virtual methods is called an interface class
- Same as Java interfaces
- Any subclass that inherits from an interface class and implements all of its methods is said to implement the interface
- Instances of the subclass may be polymorphically substituted anywhere an object of the interface type is expected
class InputStream : public ObjectCount<InputStream> {
    public:
    virtual ~InputStream() {}  
    virtual bool IsOpen() = 0;
    virtual int Read() = 0;
    virtual void Close() = 0;
};
Interface Example: InputStream

class URLConnection : public ObjectCount<URLConnection> {
public:
    static InputStream * Open(const string & url);
};

InputStream * URLConnection::Open(const string & url) {
    if (StringUtil::IsPrefix(url, "file:")) {
        return new FileInputStream(url);
    }
    else if (StringUtil::IsPrefix(url, "http://")) {
        return new HTTPInputStream(url);
    }
    else {
        throw InvalidURLException(url);
    }
}
void PrintStream(InputStream * s) {
    int c = s->Read();
    while (c != -1) {
        cout << (char)c;
        c = s->Read();
    }
}

void main(int argc, char * argv[]) {
    InputStream * doc = URLConnection::Open(argv[1]);
    PrintStream(doc);
    doc->Close();
    delete doc;
}
Virtual Methods in Chess

- You are required to use virtual methods in your Chess Program to handle the differences between the various chess pieces
- How would you do this?
Virtual Methods in Chess

class Piece : public ObjectCount<Piece> {
protected:
    ChessColor color;
    ChessDirection direction;

public:
    Piece(ChessColor c, ChessDirection d) {
        color = c;
        direction = d;
    }

    virtual ~Piece() {} 

    ChessColor GetColor() {
        return color;
    }

    virtual set<BoardPosition>
        GetCandidateMoves(Board * board, BoardPosition pos) = 0;
};
Virtual Methods in Chess

- Piece
- Pawn
- Rook
- Knight
- Bishop
- Queen
- King