

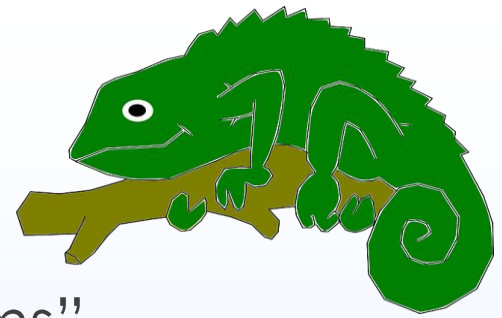
# EECS 280 – Lecture 10

## Polymorphism

1

2/20/2021

# Polymorphism



- Literally, it means “many forms”.
  - The ability for one “piece” of code to behave differently depending on how it is used.
- Ironically, there are many forms of polymorphism.
  - **Function overloading**  
(Ad-hoc Polymorphism)
  - **Templates**  
(Parametric Polymorphism)
  - **Subtype Polymorphism**

**Almost always,  
when people say  
“polymorphism”,  
they mean subtype  
polymorphism.**

# Function Overloading

- Ad-hoc Polymorphism is the use of function overloading for a **single name** to represent **many functions** within a single scope.

```
int main() {  
    Base b;  
    b.foo(42);  
    b.foo("test");  
}
```

The name foo  
can take on  
the form of  
either function.

```
class Base {  
public:  
    int x;  
    void foo(int a);  
    int foo(string b);  
    void bar(Duck *c);  
};
```

This one is separate,  
because it's in a  
different scope.

```
class Derived : public Base {  
public:  
    void x(int b);  
    int foo(int a);  
    void bar(bool c);  
};
```

# Function Overloading

- Several functions in one scope with the **same name**, but different **signatures**.
  - A function's signature includes its name and parameter types, but not return type.

```
int add(int x, int y) {  
    return x + y;  
}  
  
// REQUIRES: x and y have the same dimensions  
Matrix add(const Matrix &x, const Matrix &y) {  
    // create a return a matrix containing the  
    // element-by-element sum of x and y.  
}
```

```
int main() {  
  
    int z = add(2, 3);  
  
    Matrix a;  
    Matrix b;  
    Matrix c = add(a, b);  
}
```

# Function Overloading

5

- Several functions in one scope with the **same name**, but different **signatures**.
  - A function's signature includes its name and parameter types, but not return type.
  - **For member functions, also const vs non-const.**

```
class Matrix {  
  
    int * at(int x, int y) {  
        ...  
    }  
  
    const int * at(int x, int y) const {  
        ...  
    }  
  
}
```

```
int main() {  
  
    Matrix mat;  
    mat.at(1, 4);  
  
    const Matrix cMat;  
    cMat.at(1, 4);  
  
}
```

# Operator Overloading

- ▶ A philosophy of C++ is that user-defined types should have just as much support as built-in types.
- ▶ We can use operators (e.g. +, -, [], etc.) with our own types too!
- ▶ To do this, we just have to tell C++ what we want each operator to do.
  - ▶ The mechanism for this is **operator overloading**.

# Operator Overloading

➡ Think of an operator like a function call.

$x + y$

`operator+(x,y)`

```
int main() {  
    int x = 3;  
    int y = 3;  
    int z = x + y;  
  
    Matrix a;  
    Matrix b;  
    Matrix c = a + b;  
  
    Card c1;  
    Card c2;  
    Card c3 = c1 + c2;  
}
```

```
Matrix operator+(const Matrix &x,  
                 const Matrix &y) {  
    // create a return a matrix  
    // containing the element-by-  
    // element sum of x and y.  
}  
  
Card operator+(const Card &x,  
               const Card &y) {  
    // Adding cards doesn't actually  
    // make sense. This is just an  
    // example.  
}
```

# Insertion Operator Overloading

- ▶ To make our custom types "printable", we need them to work with the << operator.
- ▶ Just specify a **non-member** function named operator<<.

```
class Card {  
    ...  
};
```

Card.h

```
std::ostream &operator<<(std::ostream &os, const Card &s);
```

```
std::ostream &operator<<(std::ostream &os, const Card &s) {  
    // print "[rank] of [suit]" to os  
    return os;  
}
```

Card.cpp

```
int main() {  
    Card card;  
    cout << card << endl;  
}
```



```
class Pixel {
public:
    int r; int g; int b;

    Pixel(int r, int g, int b)
        : r(r), g(g), b(b) { }
};

// TASK 1: Add an overloaded operator- that
// returns the squared difference between two
// pixels (you can just call squared_difference
// in your implementation)

// TASK 2: Add an overloaded operator<< that
// prints out the pixel in this format:
//   rgb({R},{G},{B})

int main() {
    Pixel p1(174, 129, 255);
    Pixel p2(166, 226, 46);

    cout << "p1: " << p1 << endl; // p1: rgb(174,129,255)
    cout << "p2: " << p2 << endl; // p2: rgb(166,226,46)
    cout << "sq diff: " << p2 - p1 << endl; // sq diff: 531
}
```

# Subtype Polymorphism

- ▶ Subtype polymorphism allows a variable of the base type to potentially hold an object of a derived type.
- ▶ Well, almost...

```
int main() {  
    Chicken c("Myrtle");  
    Duck d("Scrooge");  
    Bird b = c;  
}
```

What's wrong  
with this code?

Chicken is 12 bytes.  
Bird is 8 bytes.



# Subtype Polymorphism

- Problem:  
Value semantics means this is trying to cram a Chicken's worth of data into the space for a Bird.<sup>1</sup>

NO

```
int main() {
    Chicken c("Myrtle");
    Bird b = c;
}
```

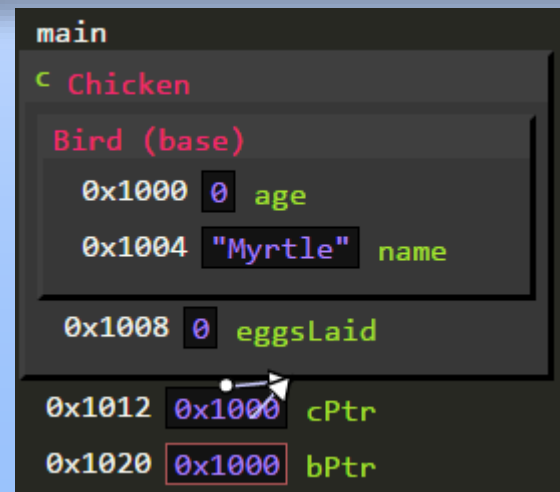
- Solution:  
Use pointers or references to work with objects indirectly.

YES

```
int main() {
    Chicken c("Myrtle");
    Chicken *cPtr = &c;
    Bird *bPtr = cPtr;
}
```

- A Bird\* and a Chicken\* take up the same amount of space! It's just an address.

<sup>1</sup> This will technically compile and run – the result is a copy of just the Bird part of the Chicken.



# Upcast vs. Downcast

- Is it safe to implicitly convert one pointer<sup>1</sup> type to another? The compiler uses this rule:
  - Upcasts are safe (and will compile)
  - Downcasts are NOT safe (and will not compile)
- For example:

```
int main() {  
    Chicken c("Myrtle");  
    Chicken *cPtr = &c;  
    Bird *bPtr = cPtr;  
}
```

**Safe. All Chickens  
are Birds.**

```
int main() {  
    Chicken c("Myrtle");  
    Bird *bPtr = &c;  
    Chicken *cPtr = bPtr;  
}
```

**NOT safe. All Birds  
are not Chickens.**

<sup>1</sup> The same rule is used for references.

## Exercise: Upcast vs. Downcast

```
int main() {  
    Bird b("Bonnie");  
    Chicken c("Carlos");  
    Duck d("Dinesh");  
}
```

### Question

Given the variables b, c, and d,  
how many of the code snippets  
on the right are safe?

- A) 0    B) 1    C) 2    D) 3    E) 4

```
Bird *bPtr = &b;  
Chicken *cPtr = bPtr;
```

```
Bird *bPtr = &b;  
bPtr = &d;  
bPtr = &c;
```

```
Bird &bRef = c;  
Chicken &cRef = bRef;
```

```
Bird &bRef = d;
```

# Exercise: Using Polymorphism

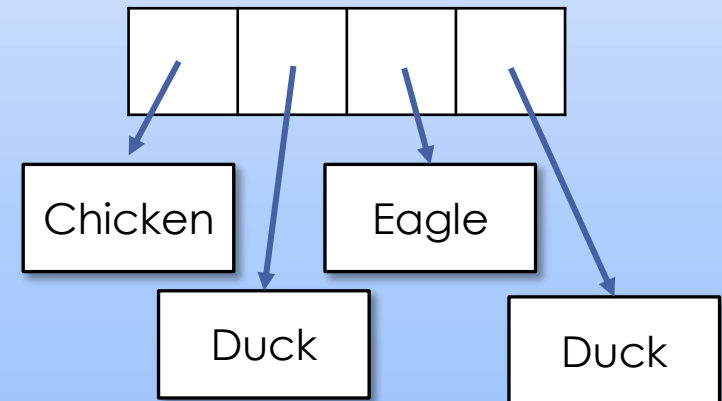
- One use case is an array of different kinds of birds.

```
void allTalk(const Bird * birds[], int length) {  
    for (int i = 0; i < length; ++i) {  
        birds[i]->talk();  
    }  
}
```

## Question

What would allTalk print  
if called on this array?

- A) bawwk, quack, screech, quack
- B) tweet, tweet, tweet, tweet
- C) bawwk, bawwk, bawwk, bawwk
- D) Undefined. The code might crash.
- E) duck, duck, goose



## Recall: Member Name Lookup

- ▶ When we use `.` or `->` for member access, how does the compiler look up the member's name?
- ▶ Start in the variable's class scope.
- ▶ If not found, try base class scope as well.
- ▶ Stop whenever a matching name is found.

```
int main() {  
    Chicken c("Myrtle");  
    c.talk();  
}
```

```
class Bird {  
    void talk() const {  
        cout << "tweet" << endl;  
    }  
};
```

```
class Chicken : public Bird {  
    void talk() const {  
        cout << "bawwk" << endl;  
    }  
};
```

# Recall: Member Name Lookup

- ▶ When we use `.` or `->` for member access, how does the compiler look up the member's name?
- ▶ Start in the variable's class scope.
- ▶ If not found, try base class scope as well.
- ▶ Stop whenever a matching name is found.

```
int main() {  
    Chicken c("Myrtle");  
    c.talk();  
  
    Bird * b = &c;  
    b->talk();  
}
```

```
class Bird {  
    void talk() const {  
        cout << "tweet" << endl;  
    }  
};
```

```
class Chicken : public Bird {  
    void talk() const {  
        cout << "bawwk" << endl;  
    }  
};
```



# Static vs. Dynamic Type

- The **static type** of a pointer/reference is the type it is declared with and is known at **compile time**.
- The **dynamic type** of a pointer/reference is the type of the object it is *actually pointing to* at **run time**.

```
1 Chicken c("Myrtle");
2 Duck d("Scrooge");
3 Bird *ptr;
4 if (random() < 0.5) {
5     ptr = &c;
6 }
7 else {
8     ptr = &d;
9 }
```

What is the static  
type of ptr here?  
**Bird\***

What is the static  
type of ptr here?  
**Bird\***

What is the dynamic  
type of ptr here?  
**undefined**

Run it. Let's say random()  
happens to return 0.3.

What is the dynamic  
type of ptr here?  
**Chicken\***

# Non-Virtual Functions

```
int main() {  
    Chicken c("Myrtle");  
    Duck d("Scrooge");  
    Bird *ptr;  
    if (random() < 0.5) {  
        ptr = &c;  
    }  
    else {  
        ptr = &d;  
    }  
    ptr->talk();  
}
```

The static type of the receiver is Bird.

tweet

What is the output of this call to talk()? Which talk() function gets used?

```
class Bird {  
    void talk() const {  
        cout << "tweet" << endl;  
    }  
};
```

```
class Duck : public Bird {  
    void talk() const {  
        cout << "quack" << endl;  
    }  
};
```


By default, member functions are non-virtual.

- For non-virtual functions, the function to call is *fixed* at compile time. This is called **static binding**.
  - Based on the **static type** of the **receiver**.

# Explicit Downcast

- ▶ A `dynamic_cast` performs an explicit downcast.<sup>1</sup>
  - ▶ If the pointed-to object is not of the requested type (or a type derived from it), the result is null.

```
int main() {  
    Chicken c("Myrtle");  
    Bird *bPtr = &c;  
    Chicken *cPtr = dynamic_cast<Chicken *>(bPtr);  
    if (cPtr != nullptr) {  
        // something chicken-specific  
    }  
}
```



Check for null

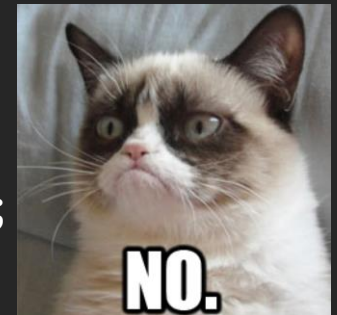
- ▶ Code that requires `dynamic_cast` usually does so as a result of poor design.
  - ▶ It can usually be restructured to use polymorphism and virtual functions. (We'll look at this in a moment...)

<sup>1</sup> C++ only allows `dynamic_cast` to be used with class types that have at least one virtual function. So this technically wouldn't work here.

# "Fixing" the allTalk Function

```
void allTalk(Bird * birds[], int length) {  
    for (int i = 0; i < length; ++i) {  
        Chicken *cPtr = dynamic_cast<Chicken*>(birds[i]);  
        if (cPtr) {  
            cPtr->talk();  
        }  
        Duck *dPtr = dynamic_cast<Duck*>(birds[i]);  
        if (dPtr) {  
            dPtr->talk();  
        }  
        Eagle *ePtr = dynamic_cast<Eagle*>(birds[i]);  
        if (ePtr) {  
            ePtr->talk();  
        }  
    }  
}
```

Do we like  
this code?



Any time we make a new  
class derived from Bird, we  
need to add another if.

# Virtual Functions

```
class Bird {
    virtual void talk() const {
        cout << "tweet" << endl;
    }
};
```

**Use virtual/override**

```
class Duck : public Bird {
    void talk() const override {
        cout << "quack" << endl;
    }
}
```

```
int main() {
    Chicken c("Myrtle");
    Duck d("Scrooge");
    Bird *ptr;
    if (random() < 0.5) {
        ptr = &c;
    }
    else {
        ptr = &d;
    }
    ptr->talk();
}
```

Run it. Let's say random() happens to return 0.7.

What is the desired output of this call to talk()? Which talk() function should be used?

The dynamic type of the receiver is Duck.

**quack**

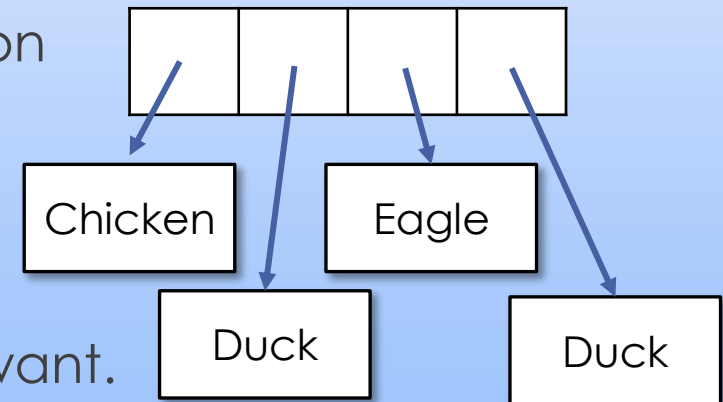
- For virtual functions, the function is not chosen until the call happens at runtime! This is called **dynamic binding**.
  - Based on the **dynamic type** of the **receiver**.

# Reprise: Using Polymorphism

- One use case is an array of different kinds of birds.

```
void allTalk(Bird * birds[], int length) {  
    for (int i = 0; i < length; ++i) {  
        birds[i]->talk();  
    }  
}
```

- Assuming the `talk` function is declared as virtual, now each call to `talk` may use the appropriate version from the derived class and print what we want.



bawwk  
quack  
screech  
quack

## Joke Time!

- ▶ How do programmers get rich?
  - ▶ Inheritance!

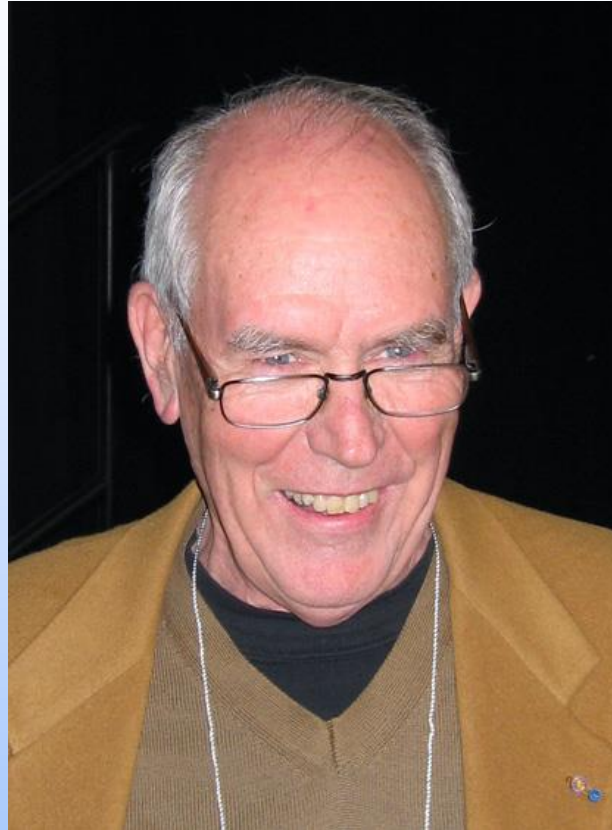
## Barbara Liskov



Foundational Work in Programming  
Language and System Design



## Ivan Sutherland



Pioneering Work in Computer Graphics  
and Object-Oriented Programming

We'll start again in one minute.



# Exercise: Virtual Functions

## Question

What does each line in main() print?

```
class Fruit {
public:
    int f1() { return 1; }
    virtual int f2() { return 2; }
};

class Citrus : public Fruit {
public:
    int f1() { return 3; }
    int f2() override { return 4; }
};

class Lemon : public Citrus {
public:
    int f1() { return 5; }
    int f2() override { return 6; }
};
```

```
int main() {
    Fruit fruit;
    Citrus citrus;
    Lemon lemon;
    Fruit *fPtr = &lemon;
    Citrus *cPtr = &citrus;

    int result = 0;
    cout << fruit.f2();
    cout << citrus.f1();
    cout << fPtr->f1();
    cout << fPtr->f2();
    cout << cPtr->f2();
    cPtr = &lemon;
    cout << cPtr->f1();
    cout << cPtr->f2();
}
```

# Overriding vs. Overloading

## ➤ Overriding

- Allowing a subclass to redefine the behavior for one of its inherited methods.



## ➤ Overloading

- A single name can refer to many different functions, depending on their parameter types.



# The override Keyword

- ▶ The `override` keyword tells the compiler to sanity check that the overriding function **signature actually matches something in the base class**.

```
class Bird {  
    virtual void talk() const { cout << "tweet" << endl; }  
};
```

Compiler allows,  
but it's probably  
a mistake!

```
class Duck : public Bird {  
    void tak() const { cout << "quack" << endl; }  
};
```

Error: no `tak()`  
function in the  
base class.

```
class Duck : public Bird {  
    void tak() const override { cout << "quack" << endl; }  
};
```

Error: Doesn't  
match the `const`  
`talk()` in base.

```
class Duck : public Bird {  
    void talk() override { cout << "quack" << endl; }  
};
```

# Awkward Bird

- Should we really have a class that is “just a bird”?
- There’s no such thing. Real world birds all belong to a more specific kind.
- The `talk()` function is awkward. There’s really no reason to pick “tweet”.
- It’s easy to imagine other functions for which the problem is worse.

```
class Bird {  
private:  
    int age;  
    string name;  
  
public:  
    Bird(string name_in)  
        : age(0), name(name_in) {  
        cout << "Bird ctor" << endl;  
    }  
    ...  
    virtual void talk() const {  
        cout << "tweet" << endl;  
    }  
  
    virtual int getWingspan() const {  
        return -1; // ???  
    }  
};
```

# Pure Virtual Functions

- Instead, we can make some functions **pure virtual**.
- This means that the definition of the function is “there is no implementation”.
- Bird is now an **abstract class**.
- You can't make any instances of an abstract class, but you wouldn't want to!
- Subclasses must<sup>1</sup> **override** the function to provide an implementation.

```
class Bird {  
private:  
    int age;  
    string name;  
  
public:  
    Bird(string name_in)  
        : age(0), name(name_in) {  
        cout << "Bird ctor" << endl;  
    }  
  
    ...  
  
    virtual void talk() const = 0;  
  
    virtual int getWingspan()  
        const = 0;  
};
```

<sup>1</sup> Otherwise, the function remains as pure virtual and the subclass is also abstract.

# Why Use Pure Virtual?

- ▶ A **virtual function** tells a subclass “You can override the behavior if you want.”
- ▶ A **pure virtual function** tells a subclass “I don’t know how to do X, so you must fill in an implementation.”
- ▶ Example: There's no sensible default value for the wingspan in our `Bird` base class from earlier. Thus, we leave the implementation of `getWingspan` to subclasses.

```
class Bird {  
public:  
...  
    virtual int getWingspan() const {  
        return -1; // ???  
    }  
};
```

Questionable

```
class Bird {  
public:  
    virtual int getWingspan()  
        const = 0;  
};  
  
int main() {  
    Bird p;  
}
```

Better

Error, Bird is an abstract class.



# Interfaces

- An **interface** is a class where all functions are pure virtual.

```
class Shape {  
public:  
    virtual double area() const = 0;  
    virtual double perimeter() const = 0;  
    virtual void scale(double s) = 0;  
};
```

- Other classes (e.g. Triangle, Rectangle) that derive from Shape are said to "implement" the Shape interface.

# Inheriting Interfaces

- ▶ We've already seen one reason to use inheritance is to save on code duplication in classes with shared functionality.
- ▶ Another, equally important use is to create hierarchies of types that all implement a base class interface.
  - ▶ Polymorphism allows functions or other code that work with the base interface to work with any object that inherits from it.
  - ▶ An example of this from the Euchre project is that your driver code only needs to worry about working with `Player*`, but it can then support any combination of simple or human players!

# Dynamic Polymorphism

- With subtype polymorphism, we can make the decision on what type to use at runtime.

Ask the user for a color.

```
int main() {  
    string color;  
    cin >> color;  
    Bird *bird = Bird_factory(color, "Myrtle");  
    bird->haveBirthday();  
    cout << "Bird " << bird->getName()  
         << " is " << bird->getAge()  
         << " and says: " << bird->talk() << endl;  
    delete bird;  
    cin >> color;  
    bird = Bird_factory(color, "Heihe");  
    cout << "Bird " << bird->getName()  
         << " is " << bird->getAge()  
         << " and says: " << bird->talk() << endl;  
    delete bird;  
}
```

This is going to return a pointer to some subclass of Bird, but we don't care what the class is called.

# The Factory Pattern

- ➡ A factory function is a function that creates and returns objects.

```
Bird * Bird_factory(const string &color,  
                    const string &name) {  
    if (color == "blue") {  
        return new BlueBird(name);  
    }  
    else if (color == "black") {  
        return new Raven(name);  
    }  
    ...  
}
```

Wait, what does new do?



# Subtypes vs. Derived Types

- ▶ Not all derived types are subtypes!
  - ▶ Anything that inherits members is a derived type.
  - ▶ A subtype must follow the is-a relationship.
- ▶ But what does the "is-a" relationship mean, exactly?
  - ▶ Liskov Substitution Principle

# Liskov Substitution Principle

- ▶ If  $S$  is a subtype of  $T$ ...
  - ▶ Any property of  $T$  should also be a property of  $S$ .
  - ▶ In any code that depends on  $T$ 's interface, an object of type  $S$  can be **substituted** without any undesirable effects.



Barbara Liskov, MIT

# Minute Exercise: Liskov Principle

Base Class	Derived Class
<pre>class Player {     // EFFECTS: Returns a card     // from the player's hand,     // following the rules of     // euchre.     Card play_card(); };</pre>	<pre>class DerivedPlayer {     // EFFECTS: Always returns the     // ace of clubs     Card play_card(); };</pre>



# Minute Exercise: Liskov Principle

Base Class	Derived Class
<pre>class PPMReader {     // EFFECTS: Reads an image from     // a stream     // REQUIRES: The stream must     // contain image data in PPM     // format. Pixels must be     // separated by a single space     // character.     Image read_ppm_image(istream &amp;is); };</pre>	<pre>class DerivedPPMReader {     // EFFECTS: Reads an image from     // a stream     // REQUIRES: The stream must     // contain image data in PPM     // format. Pixels may be     // separated by any kind of     // whitespace.     Image read_ppm_image(istream &amp;is); };</pre>

# Minute Exercise: Liskov Principle

Base Class	Derived Class
<pre>class Unicorn {     // EFFECTS: The unicorn fires a     // laser. Returns the power of the     // laser beam, which is at least     // 100kw.     double fire_laser_beam(); };</pre>	<pre>class DerivedUnicorn {     // EFFECTS: The unicorn fires a     // massive laser. Returns the power     // of the laser beam, which is over     // 9000kw.     double fire_laser_beam(); };</pre>