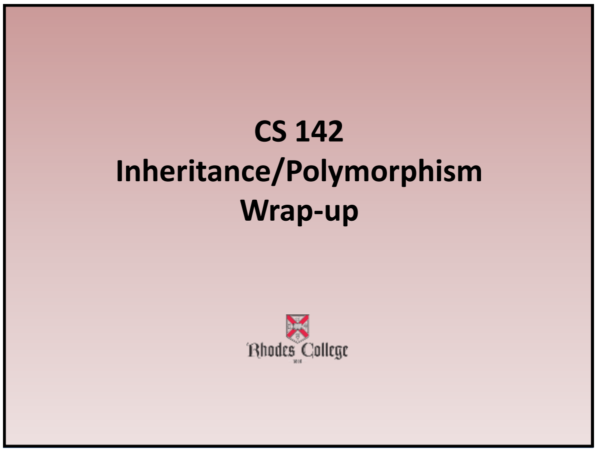
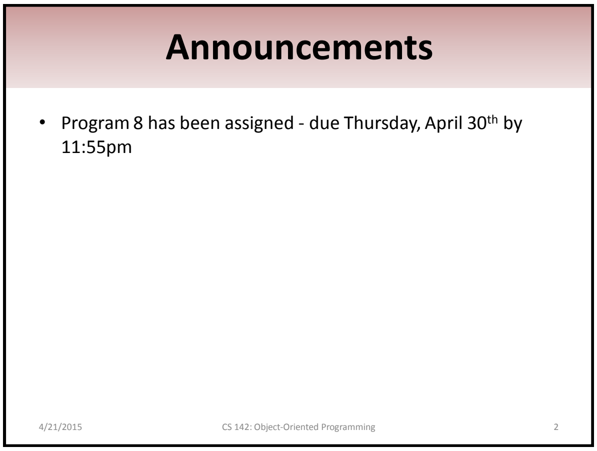


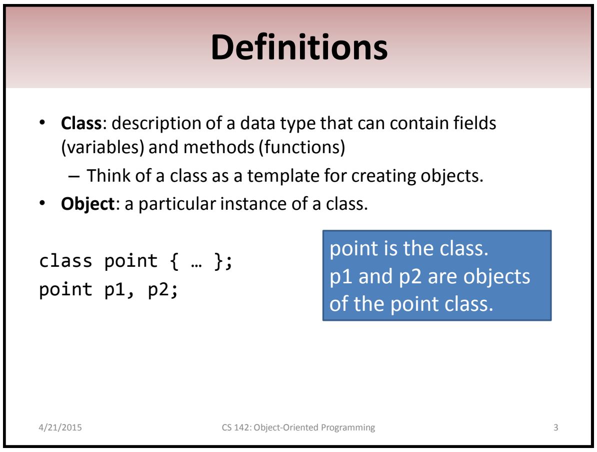
CS 142
Inheritance/Polymorphism
Wrap-up

Rhodes College



Announcements

- Program 8 has been assigned - due Thursday, April 30th by 11:55pm

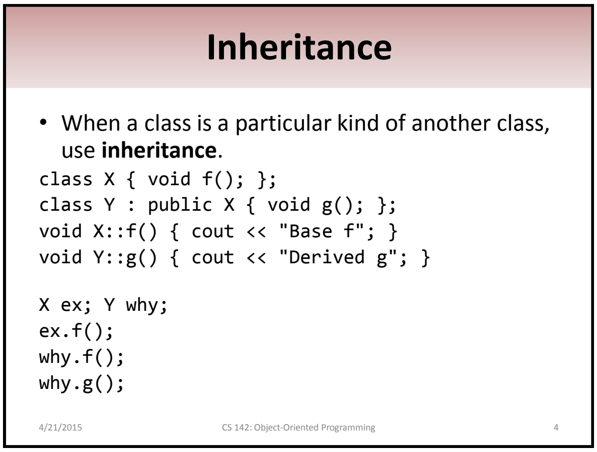


Definitions

- **Class**: description of a data type that can contain fields (variables) and methods (functions)
  - Think of a class as a template for creating objects.
- **Object**: a particular instance of a class.

```
class point { … };
point p1, p2;
```

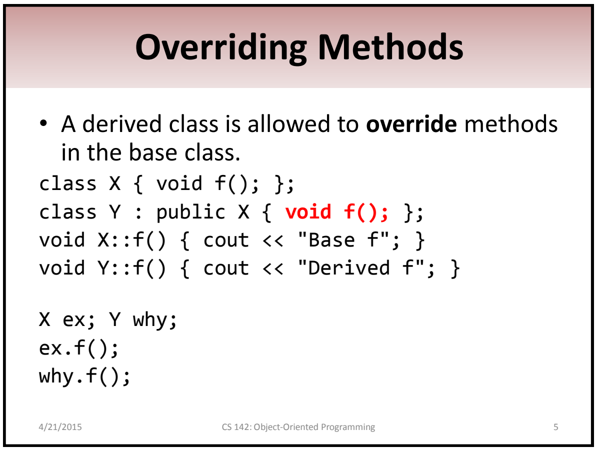
point is the class.
p1 and p2 are objects of the point class.



Inheritance

- When a class is a particular kind of another class, use **inheritance**.

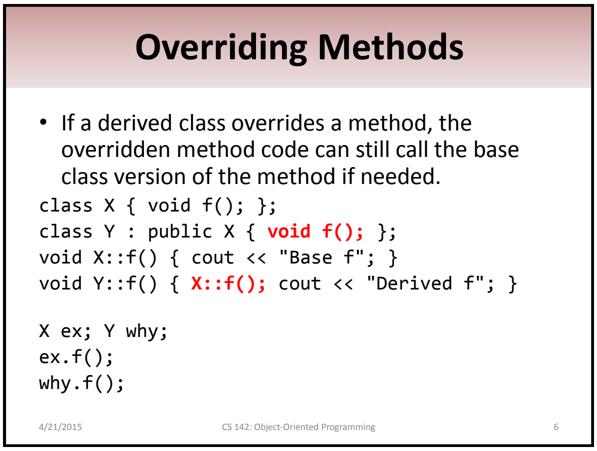
```
class X { void f(); };
class Y : public X { void g(); };
void X::f() { cout << "Base f"; }
void Y::g() { cout << "Derived g"; }

X ex; Y why;
ex.f();
why.f();
why.g();
```



Overriding Methods

- A derived class is allowed to **override** methods in the base class.

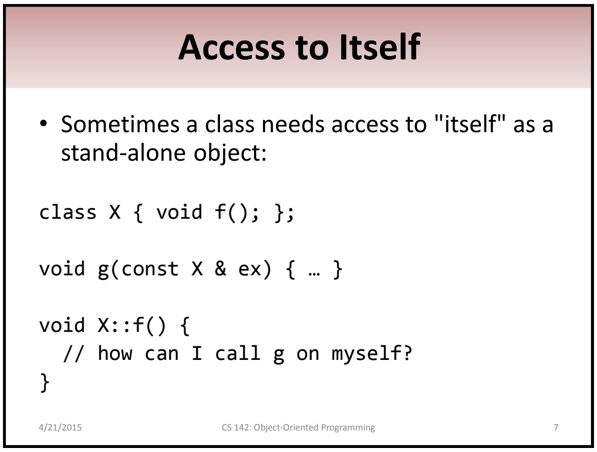
```
class X { void f(); };
class Y : public X { void f(); };
void X::f() { cout << "Base f"; }
void Y::f() { cout << "Derived f"; }

X ex; Y why;
ex.f();
why.f();
```



Overriding Methods

- If a derived class overrides a method, the overridden method code can still call the base class version of the method if needed.

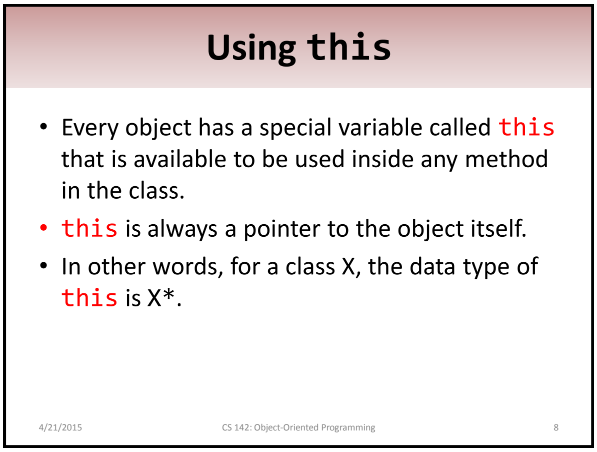
```
class X { void f(); };
class Y : public X { void f(); };
void X::f() { cout << "Base f"; }
void Y::f() { X::f(); cout << "Derived f"; }

X ex; Y why;
ex.f();
why.f();
```



Access to Itself

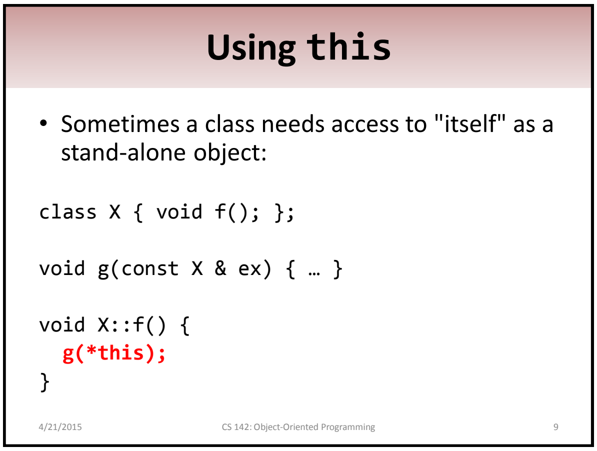
- Sometimes a class needs access to "itself" as a stand-alone object:

```
class X { void f(); };

void g(const X & ex) { … }

void X::f() {
  // how can I call g on myself?
}
```



Using `this`

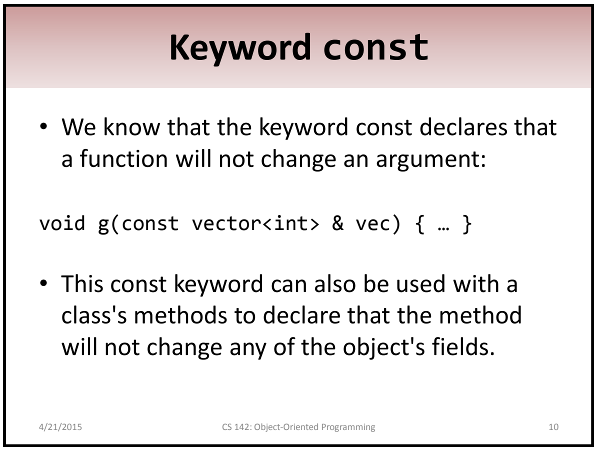
- Every object has a special variable called `this` that is available to be used inside any method in the class.
- `this` is always a pointer to the object itself.
- In other words, for a class X, the data type of `this` is X*.



Using `this`

- Sometimes a class needs access to "itself" as a stand-alone object:

```
class X { void f(); };

void g(const X & ex) { … }

void X::f() {
  g(*this);
}
```

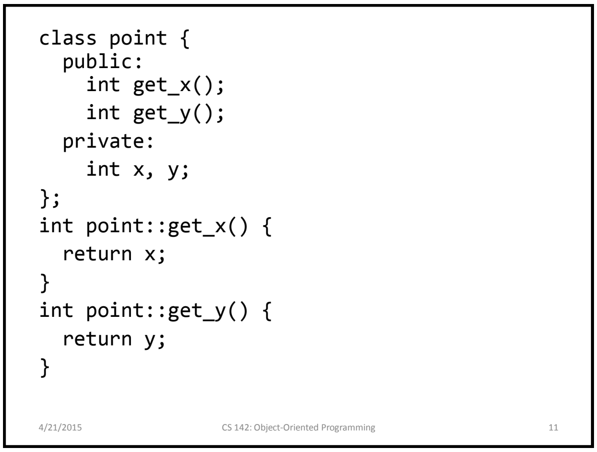


Keyword `const`

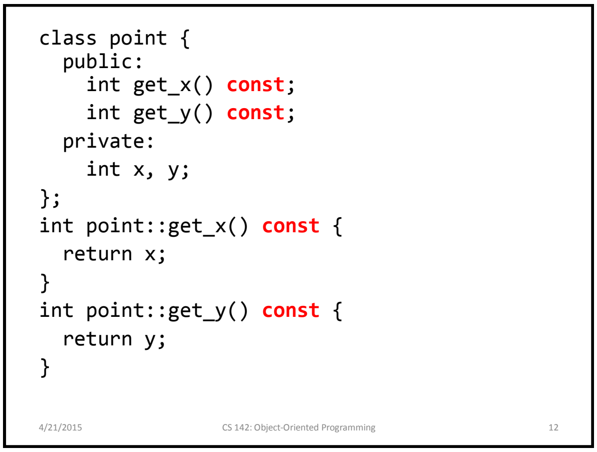
- We know that the keyword const declares that a function will not change an argument:

```
void g(const vector<int> & vec) { … }
```

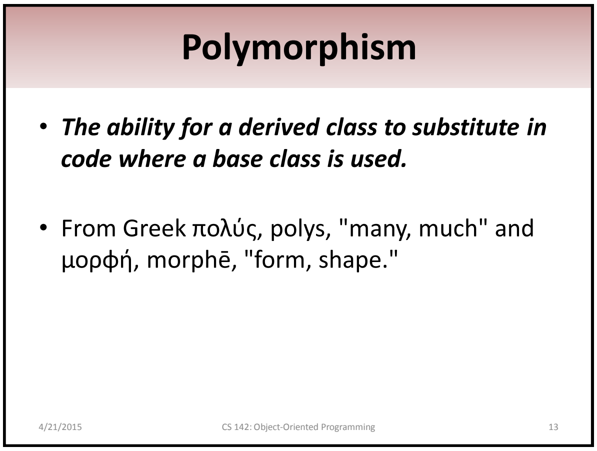
- This const keyword can also be used with a class's methods to declare that the method will not change any of the object's fields.



```
class point {
  public:
    int get_x();
    int get_y();
  private:
    int x, y;
};
int point::get_x() {
  return x;
}
int point::get_y() {
  return y;
}
```

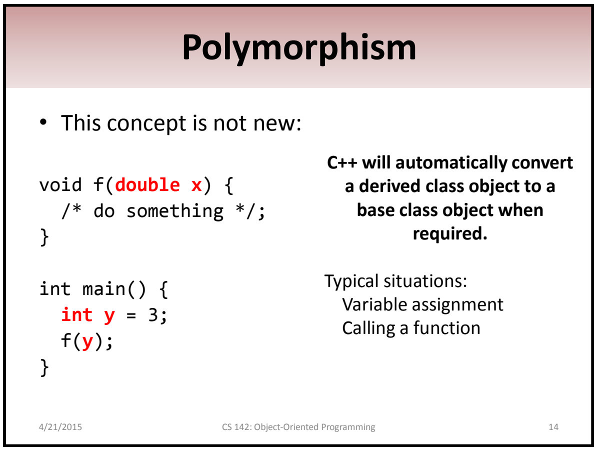


```
class point {
  public:
    int get_x() const;
    int get_y() const;
  private:
    int x, y;
};
int point::get_x() const {
  return x;
}
int point::get_y() const {
  return y;
}
```



Polymorphism

- ***The ability for a derived class to substitute in code where a base class is used.***
- From Greek πολύς, polys, "many, much" and μορφή, morphē, "form, shape."



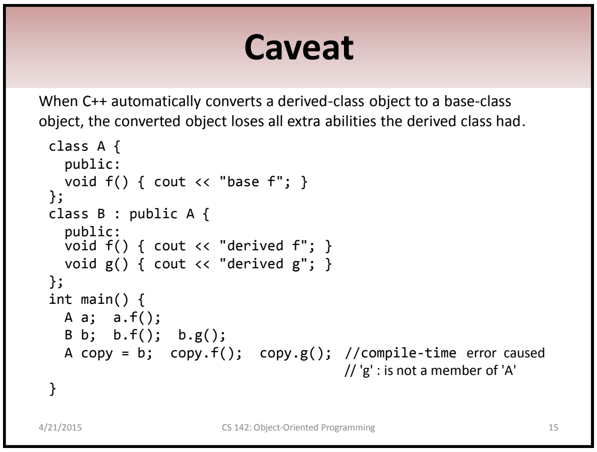
Polymorphism

- This concept is not new:

```
void f(double x) {
  /* do something */;
}

int main() {
  int y = 3;
  f(y);
}
```

C++ will automatically convert a derived class object to a base class object when required.

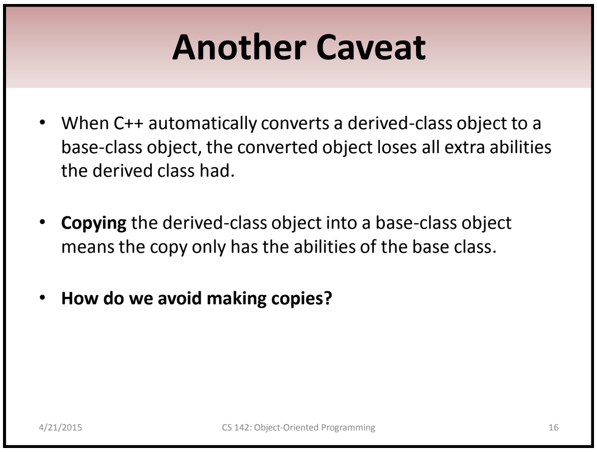
Typical situations:
  Variable assignment
  Calling a function



Caveat

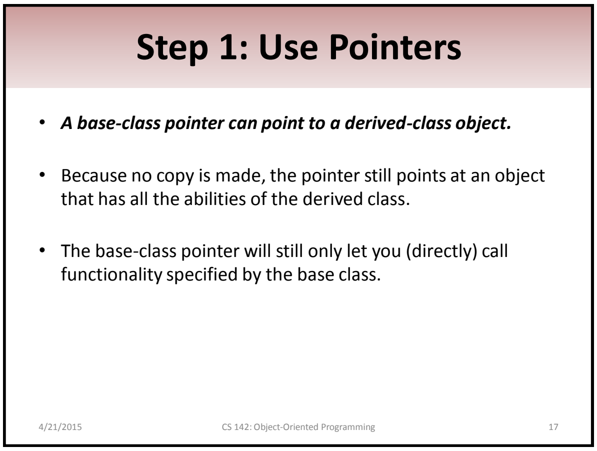
When C++ automatically converts a derived-class object to a base-class object, the converted object loses all extra abilities the derived class had.

```
class A {
  public:
  void f() { cout << "base f"; }
};
class B : public A {
  public:
  void f() { cout << "derived f"; }
  void g() { cout << "derived g"; }
};
int main() {
  A a;  a.f();
  B b;  b.f();  b.g();
  A copy = b;  copy.f();  copy.g();  //compile-time error caused
                                     // 'g' : is not a member of 'A'
}
```



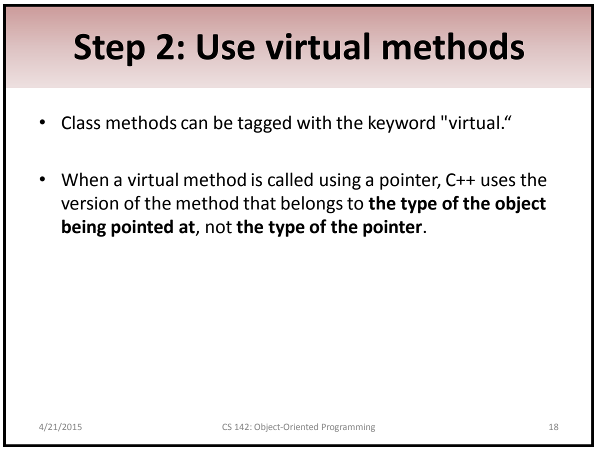
Another Caveat

- When C++ automatically converts a derived-class object to a base-class object, the converted object loses all extra abilities the derived class had.
- **Copying** the derived-class object into a base-class object means the copy only has the abilities of the base class.
- **How do we avoid making copies?**



Step 1: Use Pointers

- ***A base-class pointer can point to a derived-class object.***
- Because no copy is made, the pointer still points at an object that has all the abilities of the derived class.
- The base-class pointer will still only let you (directly) call functionality specified by the base class.



Step 2: Use virtual methods

- Class methods can be tagged with the keyword "virtual.“
- When a virtual method is called using a pointer, C++ uses the version of the method that belongs to **the type of the object being pointed at**, not **the type of the pointer**.