EECS 490 – Lecture 16

Inheritance and Polymorphism

11/2/17

Announcements

- Mid-semester survey due tomorrow at 8pm
- HW4 due Tue 11/14 at 8pm
- Project 4 due Tue 11/21 at 8pm
- Midterm regrade requests due Thu 11/9 at 8pm

Review: OOP

- *Encapsulation*: bundling together data of an ADT along with the functions that operate on the data
- *Information hiding*: restricting access to the implementation details of an ADT
- *Inheritance*: reusing code of an existing ADT when defining a new one
  - Includes *interface inheritance* and *implementation inheritance*
- *Subtype polymorphism*: using an instance of a derived ADT where a base ADT is expected
  - Requires some form of *dynamic binding*, where the derived functionality is used at runtime

The term "encapsulation" is often used to encompass information hiding as well.

OOP and Message Passing

- Conceptually, object-oriented programming consists of passing messages to objects, which then respond to the message
  - Member access on an object can be thought of as sending a message to the object

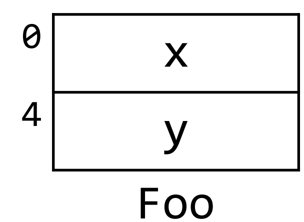
- Languages differ in:
  - Whether the set of messages an object responds to (i.e. its members) is fixed at compile time
  - Whether the actual message to be sent to an object must be known at compile time

Record[1]-Based Implementation

- In languages that prioritize efficiency, the members of an object are known at compile time
- Fields of an object are stored directly within the memory of the object, at offsets that can be computed at compile time
- Field access can be translated by the compiler to an offset into the object

```
class Foo {
public:
  int x, y;
  Foo(int x_, int y_);
};

Foo f(3, 4);
cout << (f.x + f.y);
```



[1]Records are called *structs* in C++.

Dictionary-Based Implementation

- In languages that allow members to be added to an object at runtime, an object's members are usually stored in a dictionary
  - Similar to our message-passing implementation from the notes
- A well-defined lookup process specifies how to lookup a member
  - In Python, check instance dictionary first, then class

```
class Foo:
    y = 2
    def __init__(self, x):
        self.x = x

f = Foo(3)
print(f.x, f.y, Foo.y)   # prints 3 2 2
f.y = 4
print(f.x, f.y, Foo.y)   # prints 3 4 2
```

Adds binding to instance dictionary

Slots in Python

- Python actually takes a hybrid approach, using a dictionary by default but allowing a record-like representation as well

```
class Complex(object):
    __slots__ = ('real', 'imag')
    def __init__(self, real, imag):
        self.real, self.imag = real, imag

    @property
    def magnitude(self):
        return (self.real ** 2 +
                self.imag ** 2) ** 0.5

    @property
    def angle(self):
        return math.atan2(self.imag, self.real)
```

__slots__ used to specify fields in dictionary-less objects

Objects that are dictionary-less lose the ability to add instance attributes at runtime.

Dynamic Messages

- Dictionary-based languages generally provide a means for constructing and sending a message to an object at runtime

- Example in Python:

```
>>> x = [1, 2, 3]
>>> x.__getattribute__('append')(4)
>>> x
[1, 2, 3, 4]
```

Java Reflection

- In Java, the powerful *reflection* API allows inspection of classes and objects at runtime
- Reflection can be used to construct and invoke a dynamic message

```
import java.lang.reflect.Method;

class Main {
  public static void main(String[] args)
    throws Exception {
    String s = "Hello World";
    Method m =
      String.class.getMethod("length", null);
    System.out.println(m.invoke(s)); // prints 11
  }
}
```

Types of Inheritance in C++

- C++ supports private, protected, and public inheritance
  - Determine the set of code that has access to the fact that a derived class has a specific base class
  - Most languages only support public inheritance
- Example:

```
struct A {
  void a() {
    cout << "A::a()" << endl;
  }
};

struct B : private A {
  void b() {
    A *a = this;
    a->a();
  }
};
```

B knows that A is its base class

The outside world does not

```
int main() {
  B b;
  b.b();
  b.a();         ✖
  A *ap = &b;    ✖
}
```

Abstract Methods

- A method is *abstract* if it doesn't have an implementation
  - Pure virtual functions in C++
- A class is abstract if it has at least one abstract method
- Used for interface inheritance, as well as polymorphism
- Example in Java:

Abstract class must be qualified by abstract keyword

```
abstract class A {
  abstract void foo();
}
```

Abstract method denoted by abstract keyword

Interfaces

- A class that only has abstract methods is often called an *interface*
- Java has a special mechanism for defining and implementing interfaces

```
interface I {
  void bar();
}

class C extends A implements I {
  void foo() {
    System.out.println("foo() in C");
  }
  public void bar() {
    System.out.println("bar() in C");
  }
}
```

Only one base class is allowed

Any number of interfaces can be implemented

Mixins

- Some languages decouple inheritance from polymorphism by allowing code to be inherited without establishing a parent-child relationship
- Example in Ruby:

Includes comparsion operators that call <=>

```ruby
class Counter
  include Comparable
  attr_accessor :count
  def initialize()
    @count = 0
  end
  def increment()
    @count += 1
  end
  def <=>(other)
    @count <=> other.count
  end
end
```

```
> c1 = Counter.new()
> c2 = Counter.new()
> c1.increment()
=> 1
> c1 == c2
=> false
> c1 < c2
=> false
> c1 > c2
=> true
```

Root Class

- In some languages, every object eventually derives from some root class
  - `Object` in Java, `object` in Python
- Example of code that uses the root class:

```
Vector<Object> unique(Vector<Object> items) {
  Vector<Object> result = new Vector<Object>();
  for (Object item : items) {
    if (!result.contains(item)) {
      result.add(item);
    }
  }
  return result;
}
```

Calls equals() method on item

Method Overriding

- Key to enabling subtype polymorphism
- In *static binding*, a member is looked up using the static type of a pointer or reference
  - Fields and static methods in both C++ and Java
  - Non-virtual methods in C++
- Overriding requires *dynamic binding*, where the dynamic type of an object determines which method is called
  - Non-static methods in Java
  - Virtual methods in C++
- Dynamic languages only use dynamic binding, since they don't have static types

Overriding and Overloading

- If a language supports overloading, an overriding method must have the same signature (parameter list, `const`-ness in C++) as the method it is overriding
- Example:

```
class Foo {
  int x;
  Foo(int x) {
    this.x = x;
  }
  public boolean equals(Foo other) {
    return x == other.x;
  }
}
```

Does not override `equals(Object)` method in `Object` class

```
Vector<Foo> vec = new Vector<Foo>();
vec.add(new Foo(3));
System.out.println(vec.contains(new Foo(3)));
```

Prints `false`

Override Assertion

- Java and C++ allow a method to be annotated with an assertion that it is an override, which is then checked by the compiler

```
class Foo {
  ...
  @Override
  public boolean equals(Foo other) {
    return x == other.x;
  }
}
```

Compiler detects error

- In C++:

```
virtual void foo(Bar b) override;
```

Covariant Return Types

- Some statically typed languages allow the return type of an overriding method to be a derived class of the return type of the overridden method

```
class Foo {
  int x;
  @Override
  public Foo clone() {
    Foo f = new Foo();
    f.x = x;
    return f;
  }
}
```

Overrides `Object clone()` in `Object` class

- C++ also allows covariant return types

Hidden Members

- Members that are redefined in a derived class *hide* the corresponding base class members[1]
- In Python, only methods and static fields can be hidden or overridden[2]
  - An object has a single dictionary that holds its fields
- In record-based languages (e.g. C++, Java), instance fields can also be hidden
- Most languages provide a mechanism for accessing members that are hidden or overridden
  - Common pattern in a method override is to add functionality on top of that provided by the base-class version

[1]In Java, methods in a derived class can overload those in the base class.
[2]Slots in a derived class can hide those in a base class.

Accessing Hidden/Overridden Members

- In C++, the scope-resolution operator is used to access a hidden or overridden member

```
struct A {
  void foo() {
    cout << "A::foo()" << endl;
  }
};

struct B : A {
  void foo() {
    A::foo();
    cout << "B::foo()" << endl;
  }
};
```

Call `A::foo()`

In this example, `A::foo` is hidden but not overridden, since it is non-virtual.

The super Keyword

- In many languages, including Java, the `super` keyword is used to access a base-class member

```
class A {
  void foo() {
    System.out.println("A.foo()");
  }
}

class B extends A {
  void foo() {
    super.foo();
    System.out.println("B.foo()");
  }
}
```

Python super()

- In Python, the `super()` built-in method is used to access a base-class member

```
class A:
  def foo(self):
    print('A.foo()')

class B(A) {
  def foo(self):
    super().foo()
    print('B.foo()')
```

Base-Class Constructors

- Similar syntax is used to call a base-class constructor

```
struct A {
  A(int x);
};
struct B : A {
  B(int x) : A(x) {}
};
```

Must be first item in initializer list

```
class A {
  A(int x) {}
}
class B extends A {
  B(int x) {
    super(x);
  }
}
```

Must be first statement in constructor

```
class A:
    def __init__(self, x):
        pass
class B(A):
    def __init__(self, x):
        super().__init__(x)
```

Unlike C++ and Java, Python does not insert an implicit call to a base-class constructor if one is missing.

Break Time!

- We'll start again in five minutes.

Dynamic Binding in Python

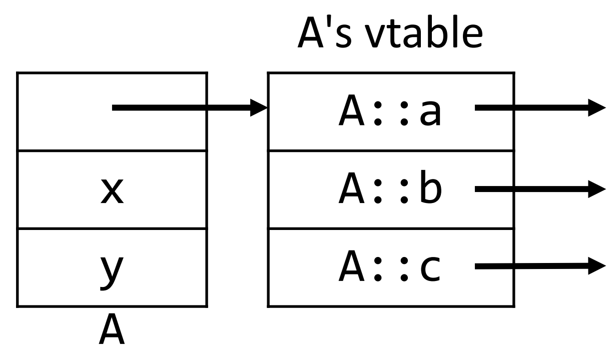
- In dictionary-based languages, dynamic binding can be implemented by a sequence of dictionary lookups at runtime

- Python lookup procedure:
  1. Check object's dictionary first
     - Instance fields stored here
  2. If not found, check the dictionary for its class
     - Static fields and all methods stored here
  3. If not found, recursively check base-class dictionaries

Virtual Tables

- In record-based implementations, a multi-step dynamic lookup process can be too inefficient
- Instead, each class has a *virtual table* (or *vtable*) that stores pointers to dynamically bound instance methods
  - Pointer to vtable stored in object
- Example:

```
struct A {
  int x;
  double y;
  virtual void a();
  virtual int b(int i);
  virtual void c(double d);
};
```



Vtables and Inheritance

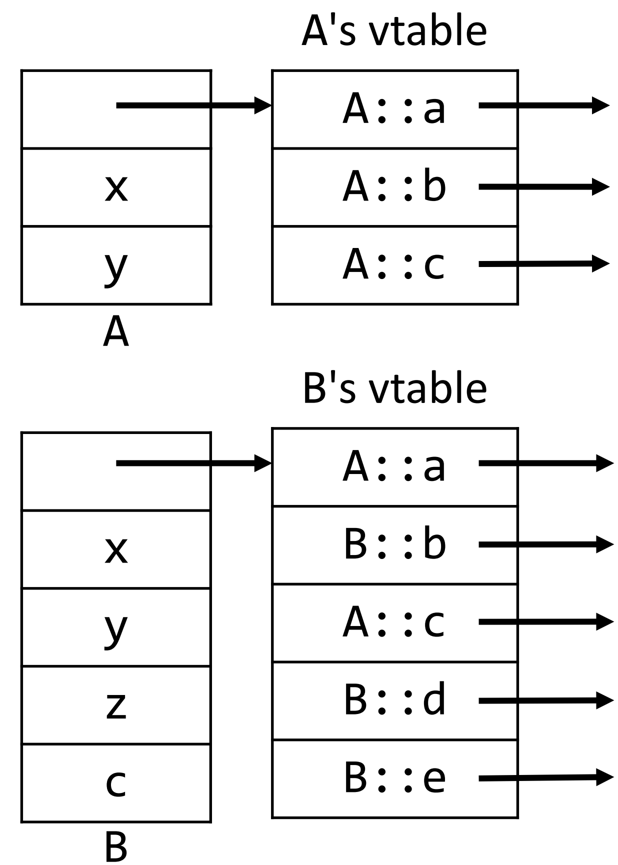
- In single inheritance, inherited instance fields and dynamically bound methods are stored at the same offsets in an object and its vtable as in the base class

```
struct B : A {
  int z;
  char c;
  virtual void d();
  virtual double e();
  virtual int b(int i);
};

A *ap = new A();
ap->x;
ap->b();
ap = new B();
ap->x;
ap->b();
```

Same offset into object

Same offset into vtable



Multiple Inheritance

- Some languages, including C++ and Python, allow a class to have multiple direct base classes

```python
class Animal:
    def defend(self):
        print('run away!')

class Insect(Animal):
    def defend(self):
        print('sting!')


class WingedAnimal(Animal):
    def defend(self):
        print('fly away!')

class Butterfly(WingedAnimal, Insect):
    pass
```

Multiple Inherited Method Definitions

- If multiple base classes define the same method, it is ambiguous which one is invoked when the method is called on the derived class

- Python uses a lookup process known as C3 *linearization*

```
>>> Butterfly().defend()
fly away!
```

- In C++, the programmer must use the scope-resolution operator to specify which method to call if it is ambiguous

```
Butterfly().WingedAnimal::defend();
```

Virtual Inheritance

- In a record-based implementation, if a base class appears multiple times, its instance fields can be shared or replicated
- Default in C++ is replication
- Virtual inheritance specifies sharing instead

```
struct Animal {
 string name;
};

struct Insect : virtual Animal {};

struct WingedAnimal : virtual Animal {};

struct Butterfly : WingedAnimal, Insect {};
```

Vtables and Multiple Inheritance

- Multiple inheritance makes it impossible to store fields and methods at consistent offsets in an object or vtable
- Instead, separate views of an object are maintained in the case of multiple inheritance, each with its own vtable

Cannot both be first entry in C

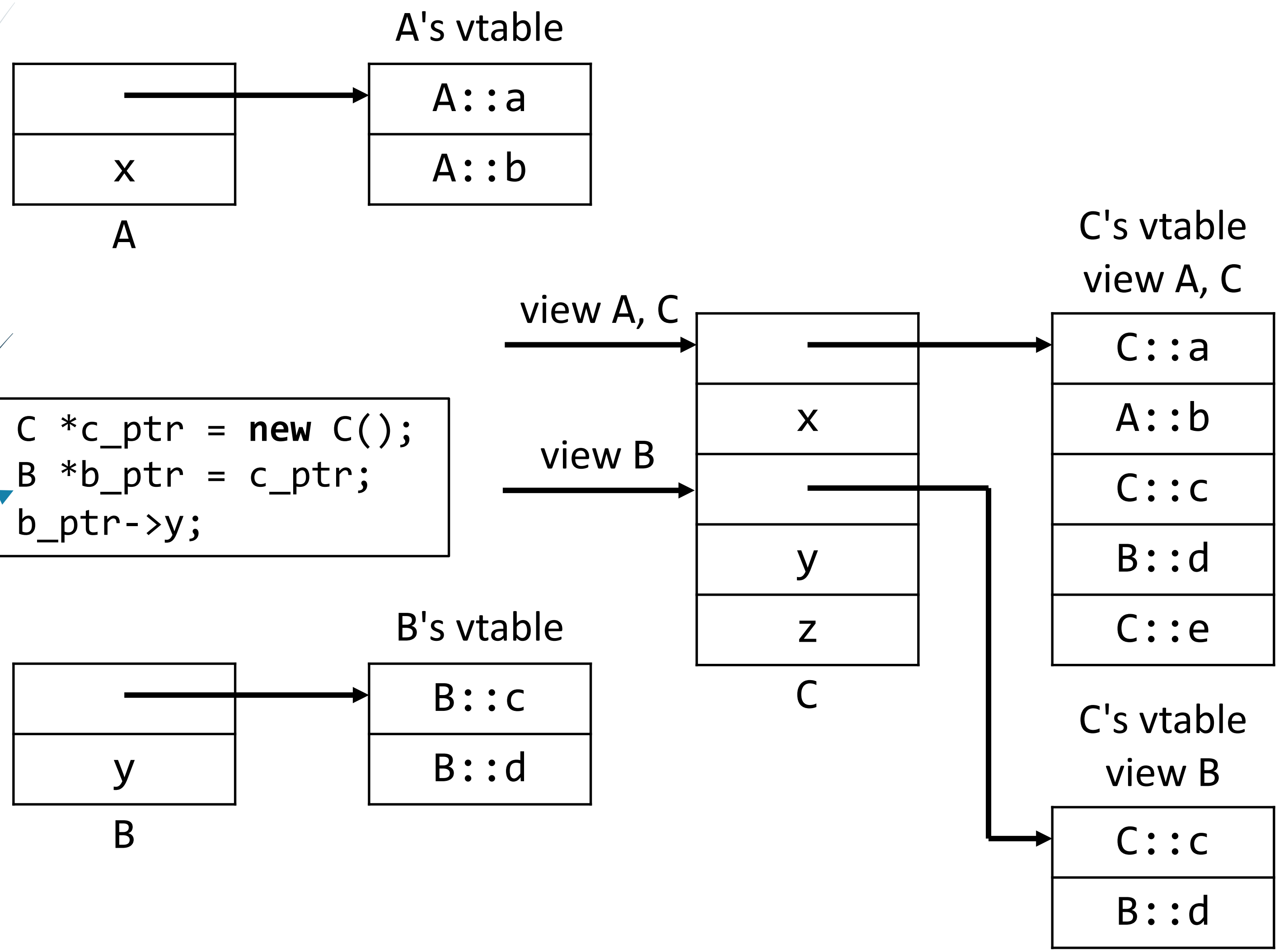
```
struct A {
  int x;
  virtual void a();
  virtual void b();
};

struct B {
  int y;
  virtual void c();
  virtual void d();
};
```

```
struct C : A, B {
  int z;
  virtual void a();
  virtual void c();
  virtual void e();
};
```

Multiple Views and Vtables

```
C *c_ptr = new C();
B *b_ptr = c_ptr;
b_ptr->y;
```



Assignment moves pointer to B view

This-Pointer Correction

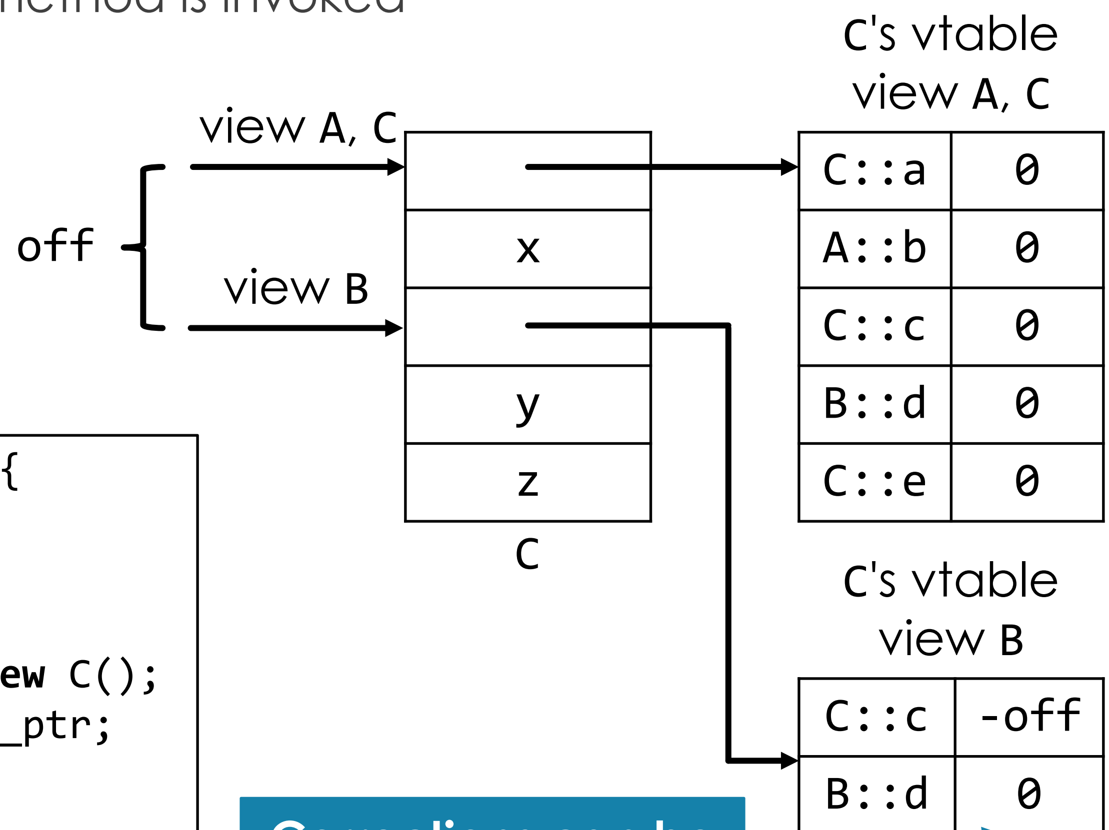
- Multiple views require a correction to the `this` pointer when a method is invoked

this pointer must be the same here

```
void C::c() {
  cout << z;
}

C *c_ptr = new C();
B *b_ptr = c_ptr;
c_ptr->c();
b_ptr->c();
```

c_ptr and b_ptr are offset by off



Corrections can be stored in vtable

In practice, a thunk is often used to perform the correction, and a pointer to the thunk is stored in the vtable.