EECS 390 – Lecture 14

Inheritance and Polymorphism

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.

Mixins

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

Includes comparsion operators that call <=>

```
> 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<>();
  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

- What does the following Java code print?

```
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

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

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 function 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

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

Must be first item in initializer list

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

Must be first statement in constructor

```python
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.

Dynamic Binding in Python

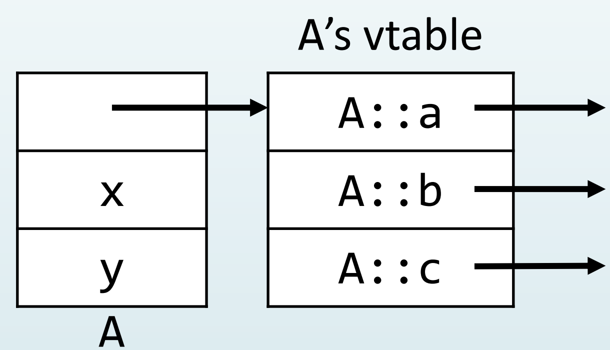
- In dictionary-based languages, dynamic binding can be implemented by a sequence of dictionary lookups at runtime

- 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);
  void f();
};
```



Statically bound by the compiler, so no pointer needed at runtime

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

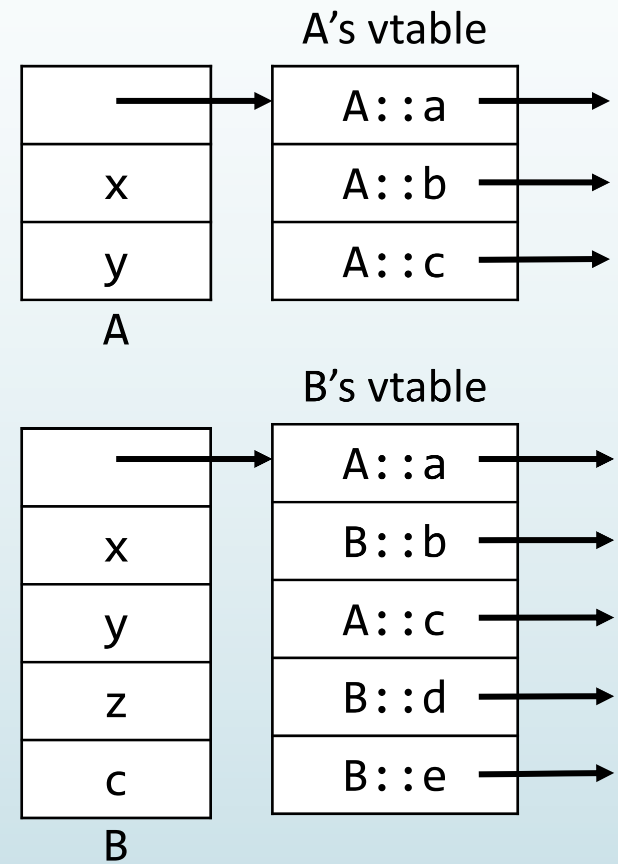
```
struct A {
  int x;
  double y;
  virtual void
    a();
  virtual int
    b(int i);
  virtual void
    c(double d);
  void f();
};
```

```
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(3);
ap = new B();
ap->x;
ap->b(3);
```

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

```
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();
```

Vtables and Multiple Inheritance

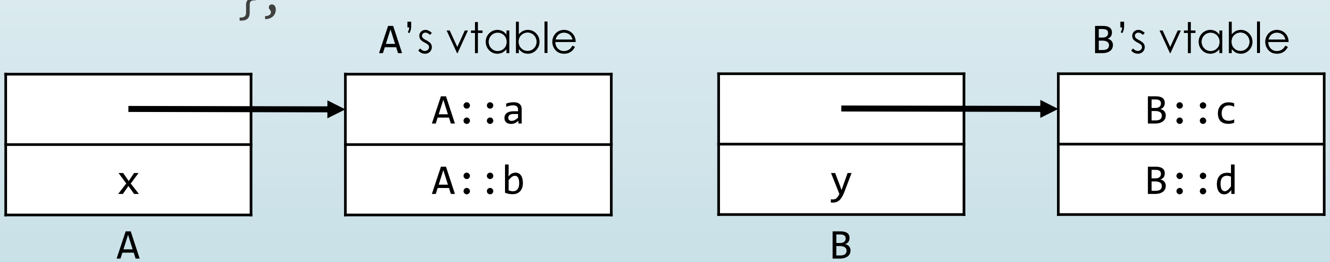
- Multiple inheritance makes it impossible to store fields and methods at consistent offsets in an object or vtable
- Instead, separate views of an object are maintained in the case of multiple inheritance, each with its own vtable

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 int d();
};
```

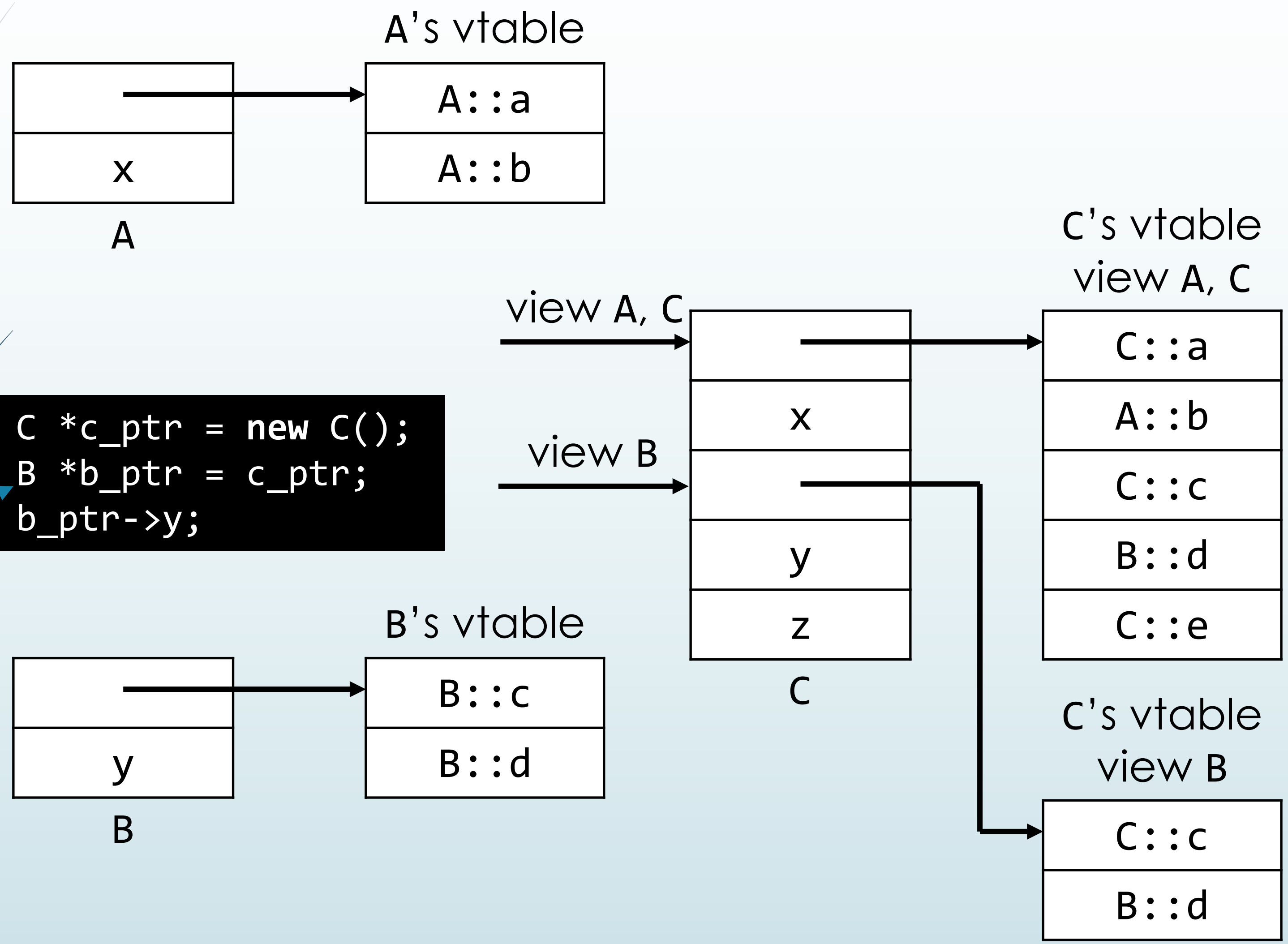
```
struct C : A, B {
  int z;
  virtual void a();
  virtual void c();
  virtual void e();
};
```



Multiple Views and Vtables

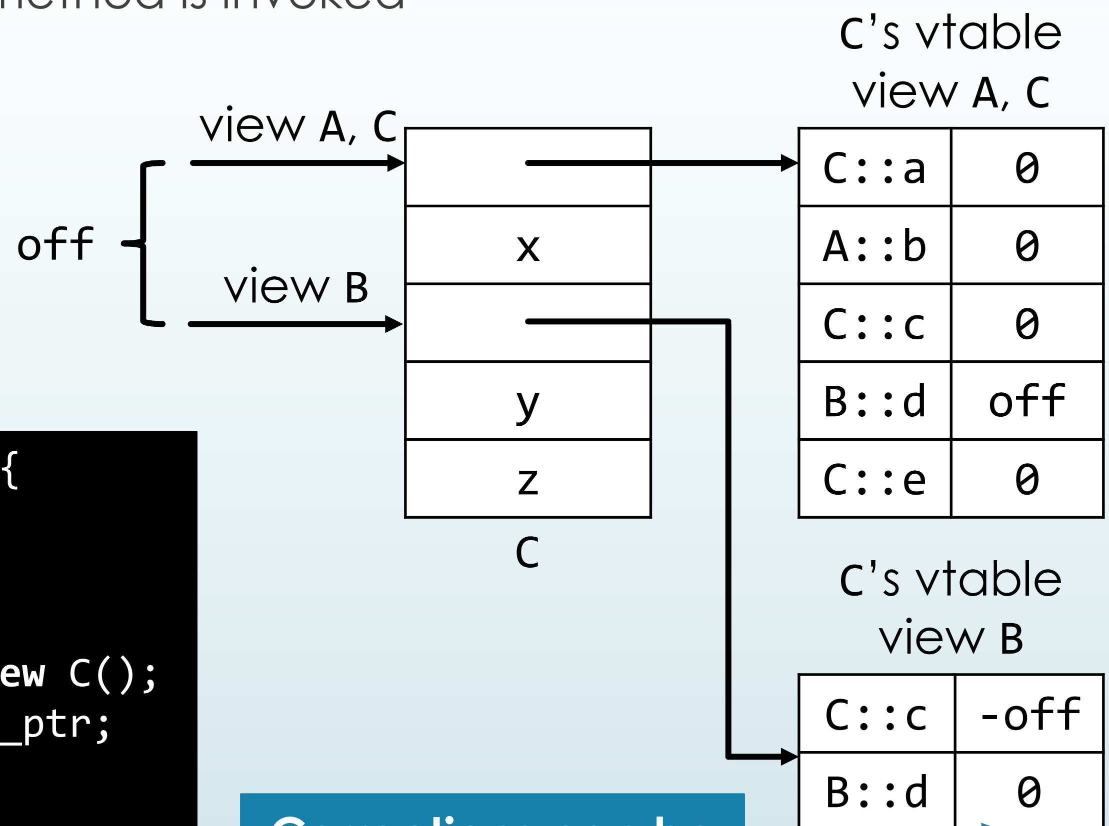
Assignment moves pointer to B view

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



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

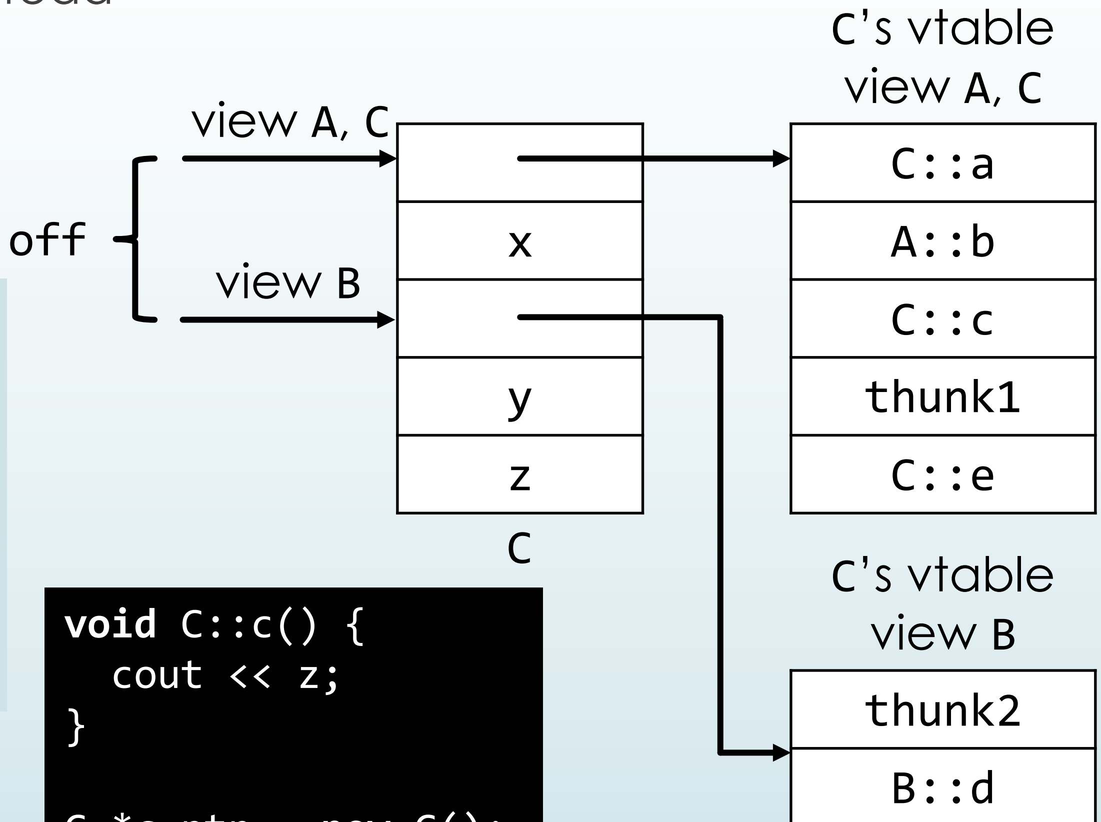
Corrections can be stored in vtable

Correction with Thunk

- Corrections can be done with compiler-generated thunks instead

```
int thunk1(C *c_ptr) {
  B *b_ptr = c_ptr;
  return b_ptr->B::d();
}
void thunk2(B *b_ptr) {
  C *c_ptr = (C*) b_ptr;
  c_ptr->C::c();
}
```

Corrects pointer




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

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

Calls thunk2

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 { int i; };
struct WingedAnimal : virtual Animal { int w; };
struct Butterfly : WingedAnimal, Insect {};
```