EECS 490 – Lecture 17

Static and Dynamic Typing

11/4/17

Announcements

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

Dynamic Binding in Python

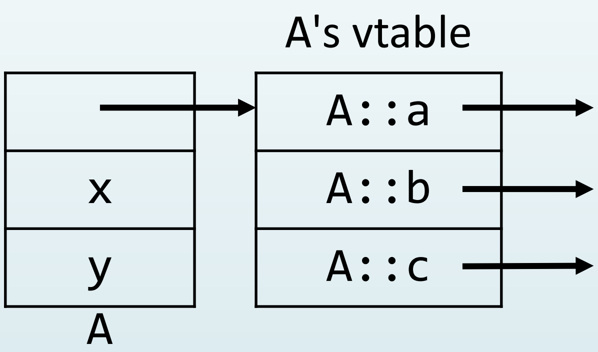
- In dictionary-based languages, dynamic binding can be implemented by a sequence of dictionary lookups at runtime

- 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

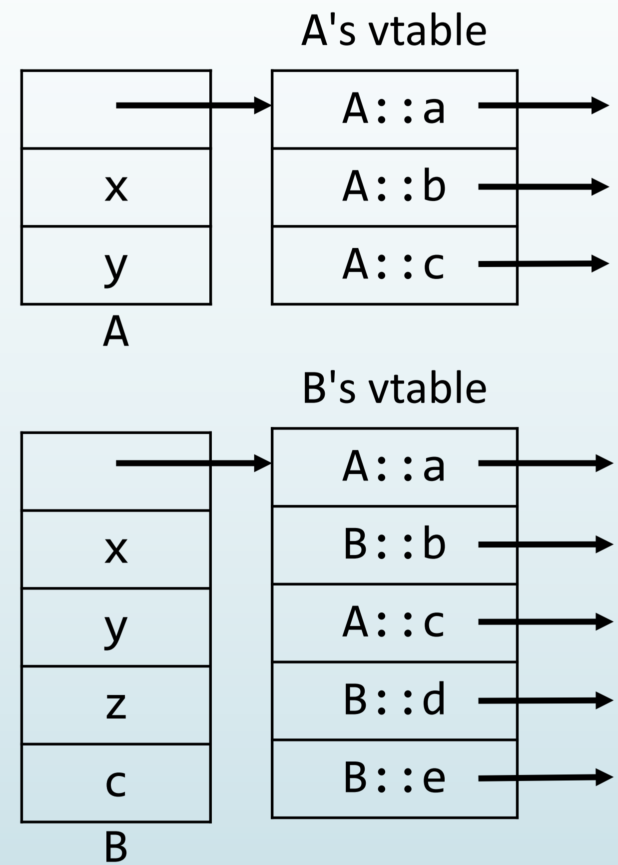
- In single inheritance, inherited instance fields and dynamically bound methods are stored at the same offsets in an object and its vtable as in the base class

```
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

```
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

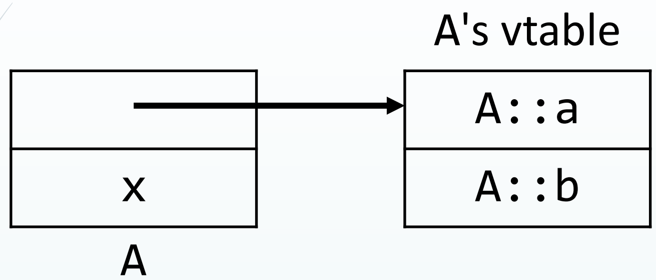
```
struct A {
  int x;
  virtual void a();
  virtual void b();
};

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

Cannot both be first entry in C

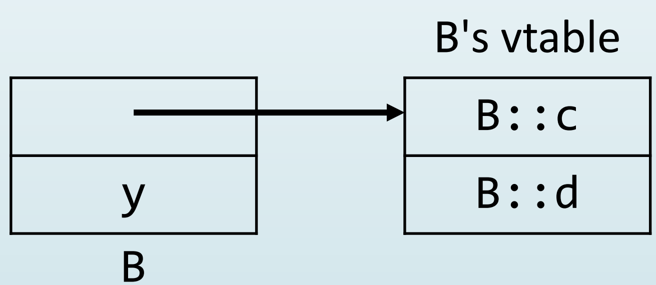
```
struct C : A, B {
  int z;
  virtual void a();
  virtual void c();
  virtual void e();
};
```

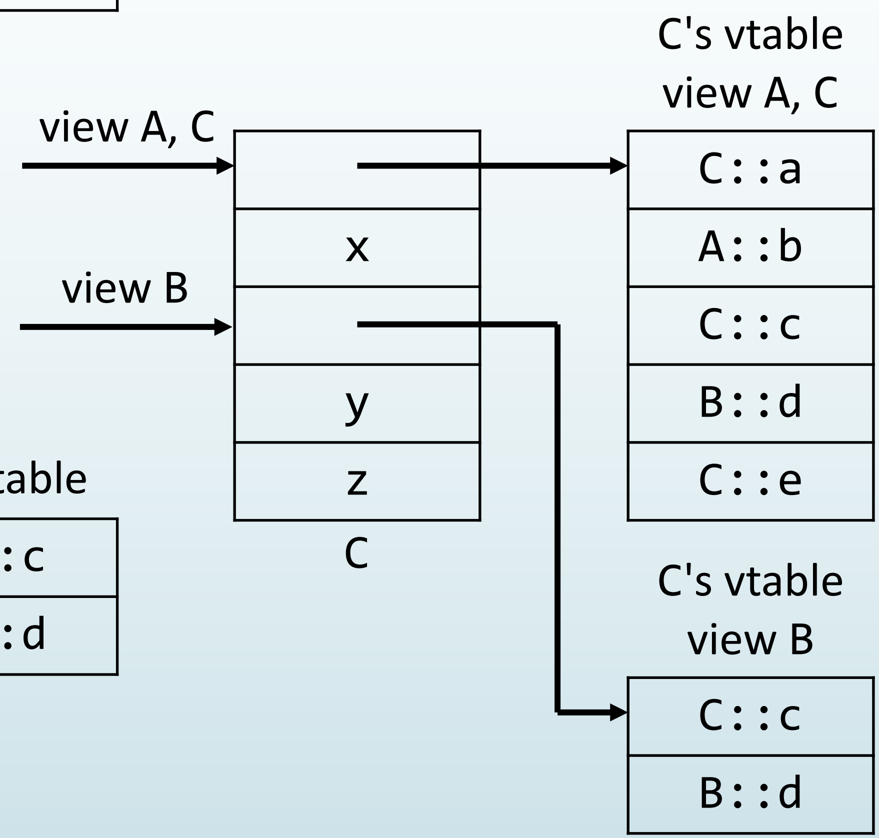
Multiple Views and Vtables



Assignment moves pointer to B view

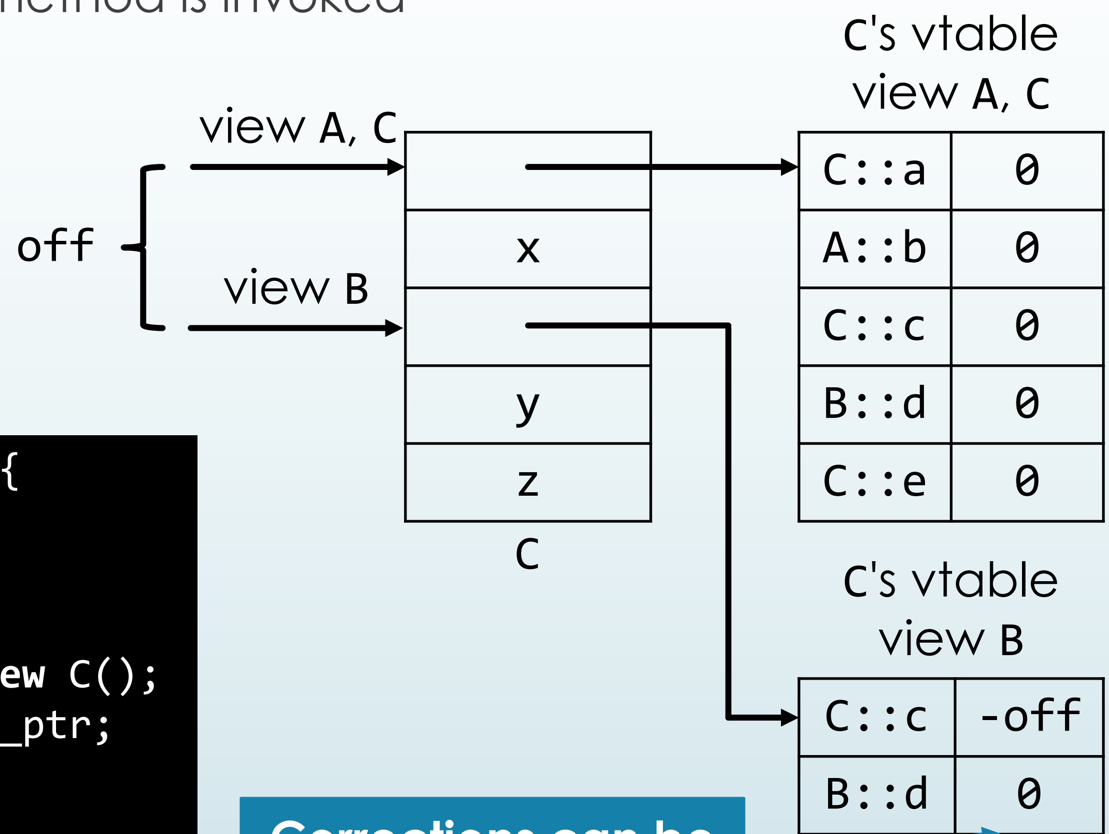
```
C *c_ptr = new C();
B *b_ptr = c_ptr;
b_ptr->y;
```





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 << x;
}

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.

Static Analysis

- Compilers perform static analysis on source code without actually running the program
- Analysis used to detect bugs and perform optimizations
- General problem of static analysis is undecidable
  - Answering any meaningful question about a program's behavior is equivalent to the halting problem
  - Instead, compilers use approximation techniques
- Type checking and control-flow analysis are two common forms of static analysis

Types

- Objects, as well as expressions, have types associated with them
  - Determine what the bits actually mean
  - Prevent common errors, such as adding a floating-point number and an array
  - Determine how operations, such as addition, are performed on the inputs
  - Serve as documentation if types are explicitly provided by the programmer
  - Allow compilers to generate specialized code
- *Type checking* ensures that types are used in semantically valid ways
  - A language is *statically typed* if this can be (mostly) done at compile time, or *dynamically typed* if it must be done at runtime

Primitive and Composite Types

- *Primitive types* are the most basic types provided by a language and are indivisible into smaller types
  - Integers, floating-point numbers, characters, pointers
- *Composite types* are composed of simpler types
  - Collections such as arrays, lists, and sets
  - *Record types* that have simpler types as *fields*
    - Structs and classes in C++

Structural Equivalence

- In some languages, two composite types are equivalent if they share the same structure

```
record A {
   int a;
   int b;
};

record B {
   int a;
   int b;
};

A x;
B y = x;
```

In a few languages (e.g. ML), order of fields does not matter

Allowed since A and B have the same structure

Name Equivalence

- Most languages distinguish between types that have different textual definitions

```
A x;
B y = x;
```

Erroneous in name equivalence

- In *strict name equivalence*, aliases are considered distinct types

- In *loose name equivalence*, aliases are the same type

```
typedef double weight;
using height = double;
height h = weight(200.);
```

Allowed in loose equivalence, forbidden in strict

Type Compatibility

- Type checking doesn't generally require type equivalence, but rather that the type used in a context is *compatible* with the expected type

- Subtype polymorphism is one example: a derived type can be used where a base type is expected

- Languages often allow a type to be implicitly converted, or *coerced*, to the expected type in certain contexts
  - Example: l-value to r-value conversion
  - Also commonly used for built-in numeric types

Type Coercion

- Operations between different types
  - For numeric types, *promotion* rules specify which types are converted to other types

```
int x = 3;
double y = 3.4;
cout << (y + x) << endl;  // result is 6.4
```

Promoted to double

- Initialization and assignment (including argument-to-parameter initialization in function calls)

```
int x = 3.4;
double y = 3;
```

OK in C++, error in Java

OK in both C++ and Java

- Some languages, such as C++, allow user-defined implicit conversions

Type Qualifiers

- Coercion rules specify how type qualifiers are allowed to be implicitly modified
- Example: `const` in C++

```
int a = 3;
const int b = a;  // OK: l-value to r-value
a = b;            // OK: const l-value to
                          r-value
int &c = a;       // OK: no coercion
int &d = b;       // ERROR: const l-value to
                             non-const l-value
const int &e = a; // OK: non-const l-value to
                          const l-value
```

Break Time!

- We'll start again in five minutes.

Types of Expressions

- Types must be determined for every expression

```
int main() {
  cout << ("Weight is " + to_string(10) +
           " grams") << endl;
}
```

String concatenation

Stream insertion

- Types of arguments used for function overload resolution
- Type of function call is return type of function
- Type of result of built-in operator defined by language according to operand types

Conditional Expression

- Non-trivial to determine type of conditional expression when the types of the last two operands differ

```
int x = 3;
double y = 3.4;
rand() < RAND_MAX / 2 ? x : x + 1;
rand() < RAND_MAX / 2 ? x : y;
```

Both operands are ints, so the result is int

Result is double

- C++ has complex conversion rules that are specific to conditional expressions
  - Type of exactly one of the two operands must be convertible to the other under a restricted set of allowed conversions

Type Inference

- Compiler must infer types of intermediate expressions, since their types are not provided by the programmer
- Some languages allow types to be elided in other contexts, if the type can be unambiguously deduced

```
int main() {
  auto func = [](int x) {
    return x + 1;
  };
  cout << func(1) << endl;
}
```

Explicitly request type deduction

Return type of lambda inferred from return expression

The `decltype` Keyword

- In C++, a variable declared with `auto` requires an initializer from which the type can be deduced

- In some contexts, an initializer cannot be provided, so `decltype` can be used instead

```
template<typename T, typename U>
class Foo {
  T a;
  U b;
  decltype(a + b) c;
};
```

Request type of expression `a + b`

Control-Flow Analysis

- The control flow of a program can also be analyzed to find potential errors
- Examples: uninitialized variables, missing return statements, and unreachable code
- Compile-time analysis must make approximations

```
if (x > 0) {
   ...
}
if (x <= 0) {
   ...
}
```

Most compilers cannot guarantee that exactly one test succeeds

```
if (x > 0) {
   ...
} else {
   ...
}
```

Exactly one branch must run

Uninitialized Variables

- Some languages (e.g. Java) consider it an error if a control path exists such that the compiler cannot guarantee that a variable is initialized before use

```
class foo {
  public static void main(String[] args) {
    int i;
    if (args.length > 0) {
      i = args.length;
    }
    if (args.length <= 0) {
      i = 0;
    }
    System.out.println(i);
  }
}
```

```
foo.java:10: error: variable i might not have been initialized
    System.out.println(i);
                       ^
1 error
```

Function Return

- Control-flow analysis can also be used to determine if there is a control path through a function that will not reach a return statement

```
static int bar(int x) {
  if (x > 0) {
    return 1;
  }
  if (x <= 0) {
    return 0;
  }
}
```

```
bar.java:9: error: missing return statement
  }
  ^
1 error
```

```
bar.cpp:12:1: warning: control may reach end of non-void function
      [-Wreturn-type]
}
^
1 warning generated.
```

Unreachable Code

- In some languages, such as Java, it is an error for there to provably be no control path that reaches a statement

```
static int baz(int x) {
  while (true) {
    if (x < 0) {
      return 0;
    }
  }
  return 1;
}
```

Compiler can determine that the return is the only exit from the loop

Unreachable code

```
baz.java:8: error: unreachable statement
    return 1;
    ^
1 error
```

Duck Typing

- Languages that do not have static typing are often implicitly polymorphic

- An object can be used in a context that requires a duck if it looks like a duck and quacks like a duck

- Example:

```
def max(x, y):
    return x if x > y else y
```

- A downside is that duck typing depends only on the name of the operation
  - Example: `run()` on an `Athlete` may have it start a marathon, while on a `Thread` it may have it start executing code

Runtime Type Information (RTTI)

- Many languages make some amount of dynamic type information available to the programmer at runtime
- Example: check if an object is an instance of a given type
  - C++: `dynamic_cast`
  - Java: `instanceof`
  - Python: built-in `isinstance()` function
- Example: obtain a representation of the type of an object at runtime
  - C++: `typeid`
  - Java: `getClass()` method on all objects
  - Python: built-in `type()` function

C++ dynamic_cast

- Attempts to cast a pointer (or reference) to a pointer (or reference) of another type
- The types must be *polymorphic*, meaning they define at least one virtual function
  - Can then use vtable pointers or entries to check cast
- Example:

```
struct A {
  virtual void bar() {
  }
};

struct B : A {
};
```

Produces null upon failure

```
void foo(A *a) {
  if (dynamic_cast<B *>(a)) {
    cout << "got a B" << endl;
  } else {
    cout << "not a B" << endl;
  }
}
```

References can't be null, so a failed cast on references throws an exception.

C++ typeid

- C++ has a `typeid` operation, which resides in the `<typeinfo>` header
- Works on values of any type, as well as types themselves
- Produces a reference to an instance of `std::type_info`, which contains basic information about the type

```
int main() {
  const type_info &i1 = typeid(int);
  const type_info &i2 = typeid(new A());
  const type_info &i3 = typeid(main);
  cout << i1.name() << " " << i2.name()
       << " " << i3.name() << endl;
}
```

Name is implementation-dependent

Prints `i P1A FivE` **on Clang**

Arrays in Java

- Java arrays are subtype polymorphic
  - If `B` derives from `A`, then `B[]` derives from `A[]`
- This allows methods to be defined that can operate on any array that holds object types
- However, it enables Bad Things to happen:

```
String[] sarray = new String[] { "foo", "bar" };
Object[] oarray = sarray;
oarray[1] = new Integer(3);
sarray[1].length();
```

Uh-oh

OK, since `String[]` derives from `Object[]`

OK from the point of view of the type system since an `Object[]` can hold an `Integer`

- To avoid this, Java checks when an item is stored in an array and throws an `ArrayStoreException` if the dynamic types are incompatible