# EECS 390 – Lecture 15

## Static and Dynamic Typing



3/17/24

# Review: 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

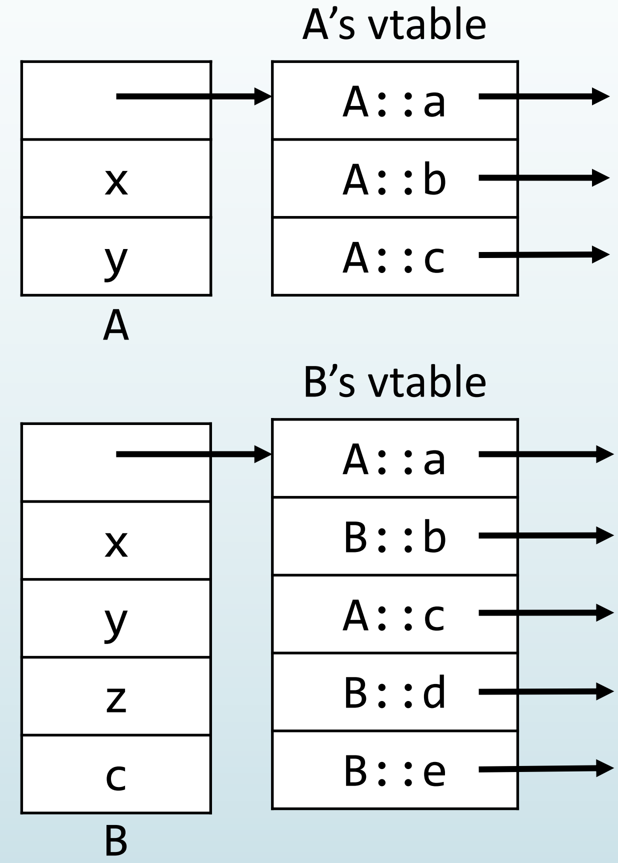
```
struct A {
  int x;
  double y;
  virtual void
    a();
  virtual int
    b(int i);
  virtual void
    c(double d);
  void f();
};
```

```
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

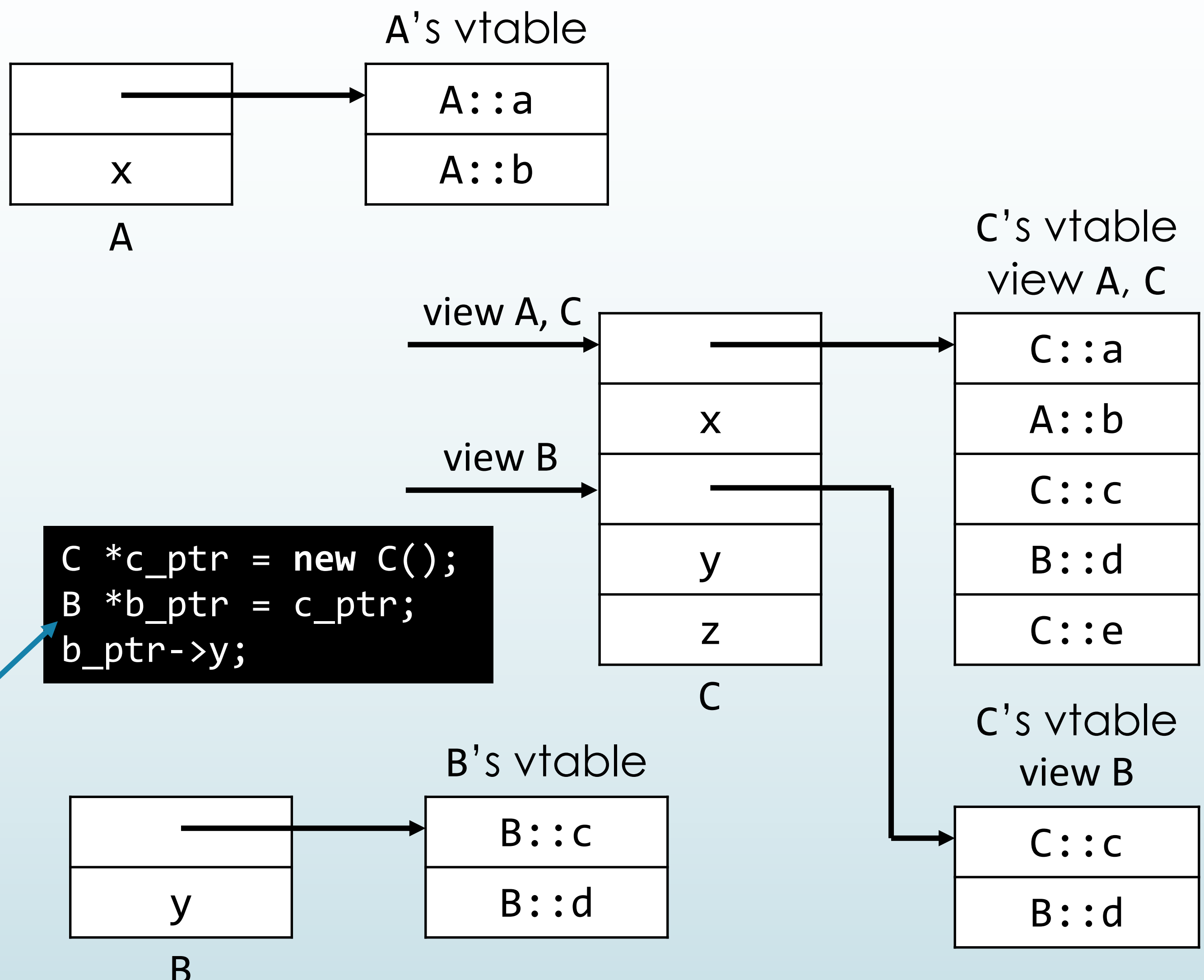


# Review: Multiple Inheritance

```
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;
  void a() override;
  void c() override;
  virtual void e();
};
```

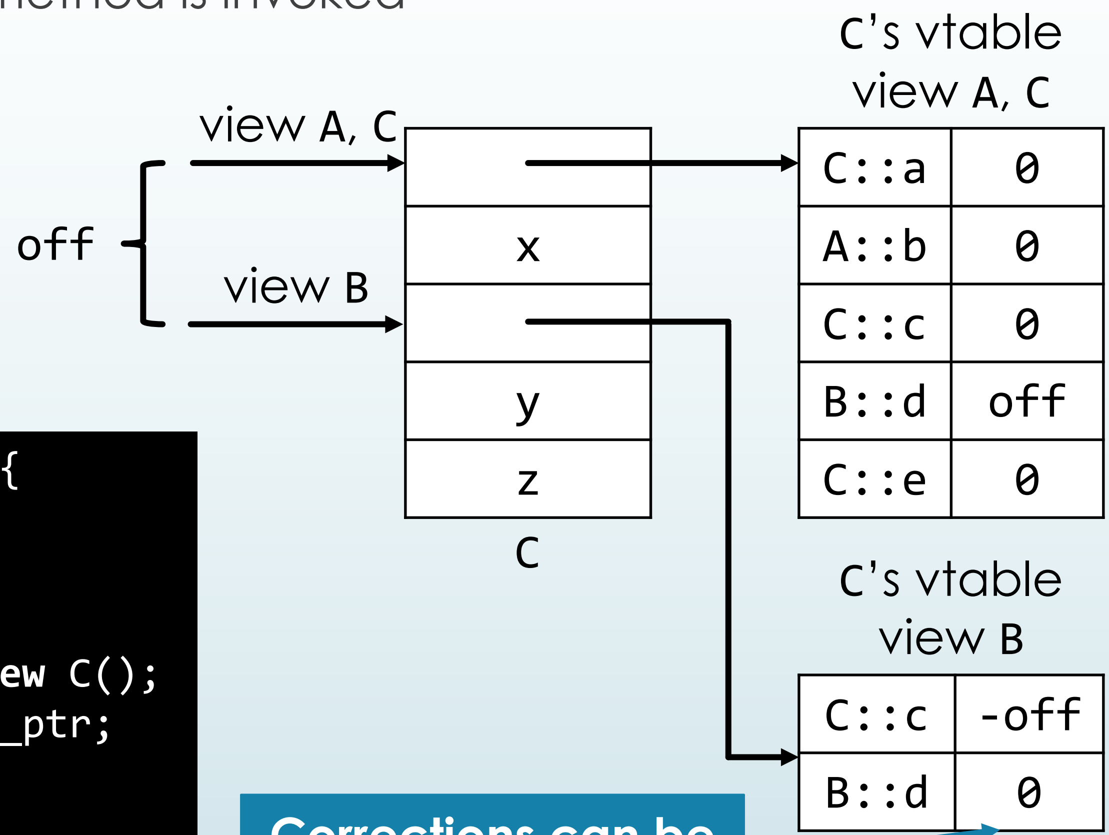
```
C *c_ptr = new C();
B *b_ptr = c_ptr;
b_ptr->y;
```

**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**

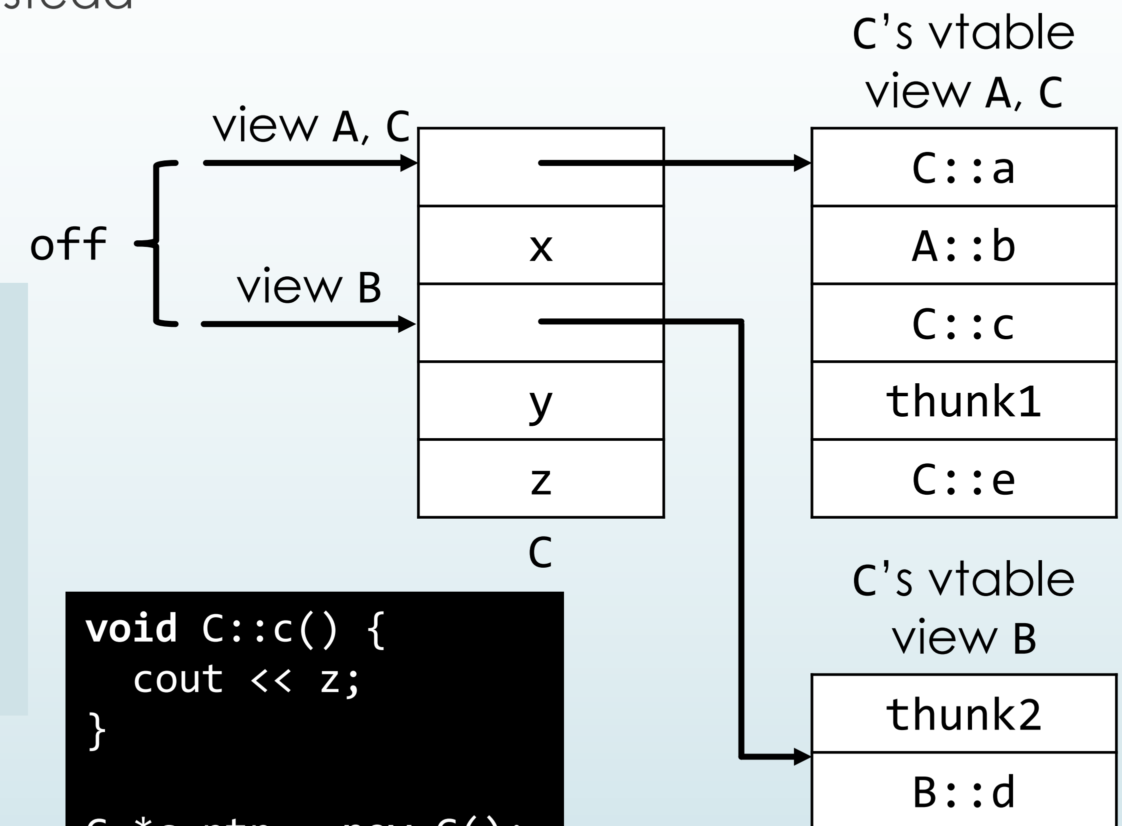
```
void C::c() {
  cout << z;
}

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**

# 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

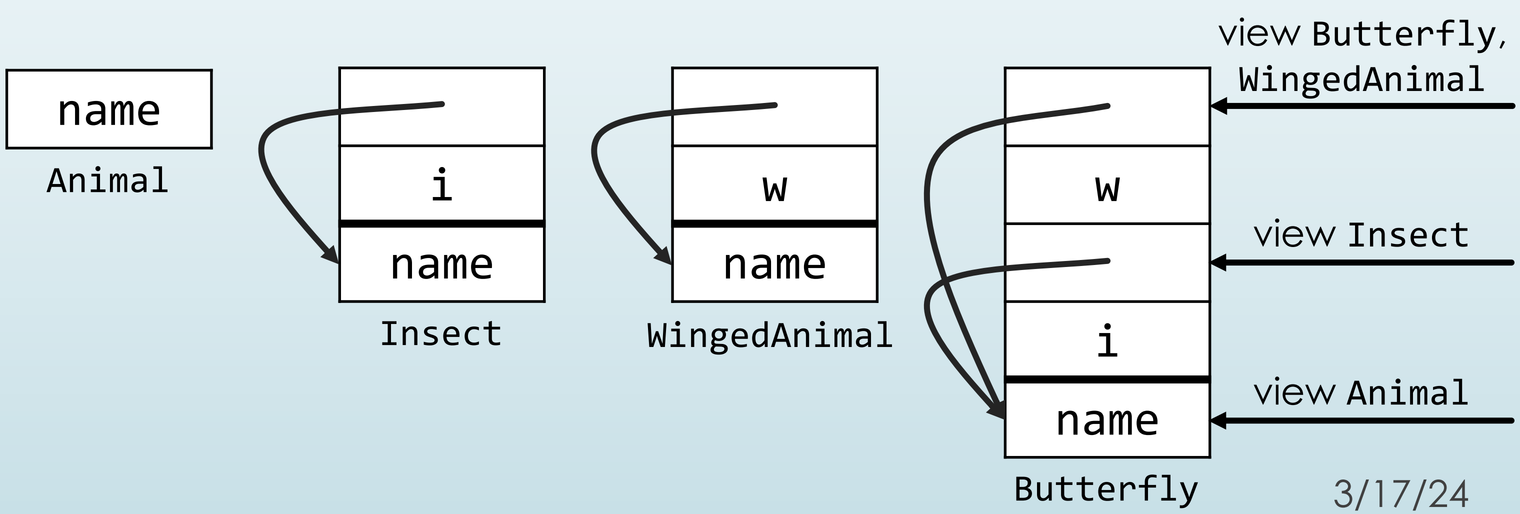
```
void C::c() {
  cout << z;
}

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 {};
```



# 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 does not 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
```

# 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

# 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`**

# 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