`dynamic_cast` from scratch

Two kinds of inheritance graphs

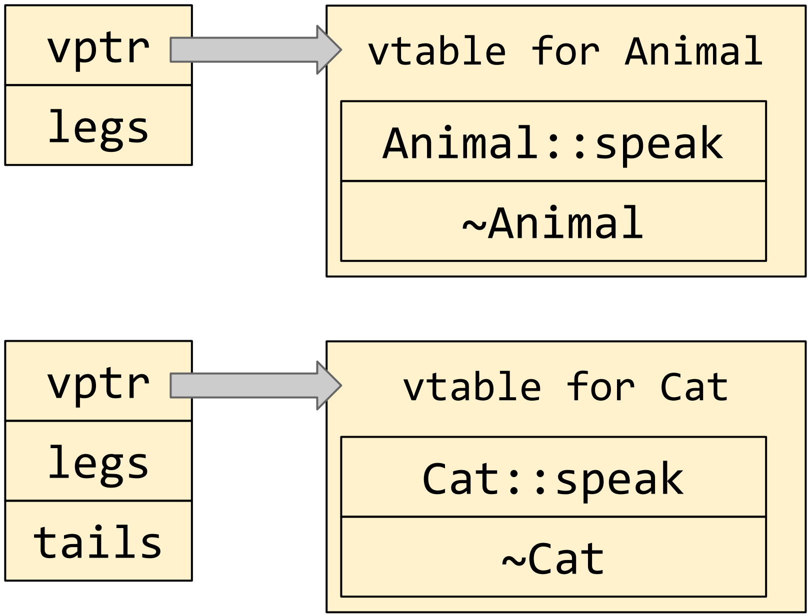
Arthur O’Dwyer
2017-07-28

Outline

- **Part I: Recap polymorphism**
  - Inheritance, vtables, and the Dreaded Diamond [3–10]
  - Accessing members of virtual bases [11–21]
  - How to visualize complex class layouts [22–29]
- Part II: `dynamic_cast`
  - What should `dynamic_cast` do? [31–54]
  - Okay, how do we implement that? [55–60]
  - Benchmark numbers [61–64]

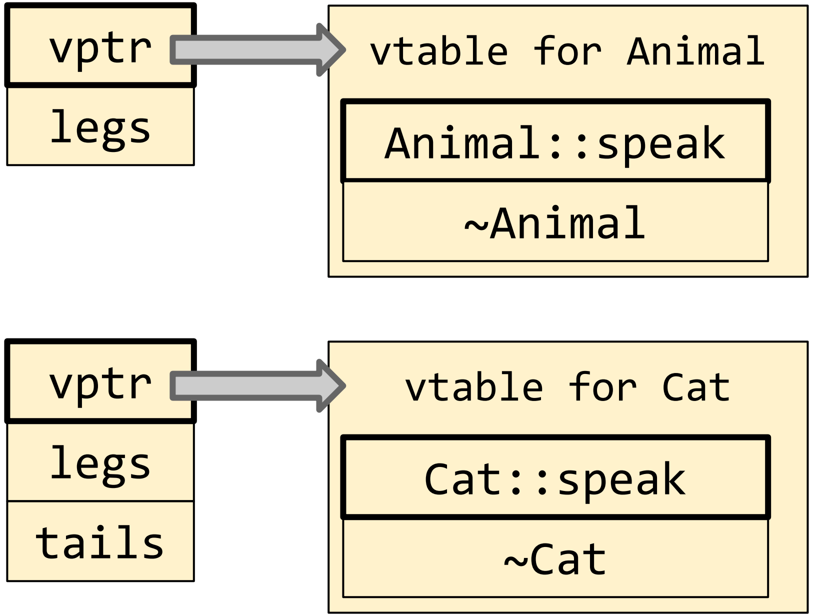
Quick recap: Polymorphism

```
class Animal {
    int legs;
    virtual void speak() { puts("hi"); }
    virtual ~Animal();
};

class Cat : public Animal {
    int tails;
    void speak() override {
        printf("Ouch, my %d tails!",
            tails);
    }
};
```



Quick recap: Polymorphism

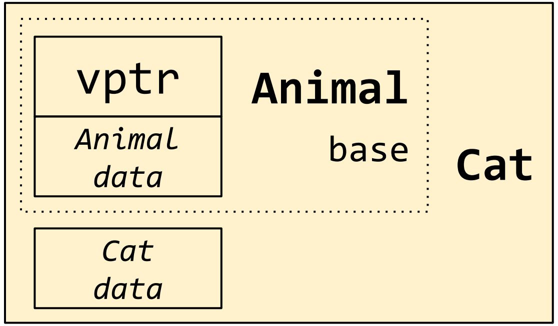
```
class Animal {
    int legs;
    virtual void speak() { puts("hi"); }
    virtual ~Animal();
};

class Cat : public Animal {
    int tails;
    void speak() override {
        printf("Ouch, my %d tails!",
            tails);
    }
};
```

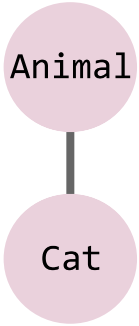



```
# a->speak();
movq  (%rdi), %rax
callq *(%rax)
```

Several graphical representations

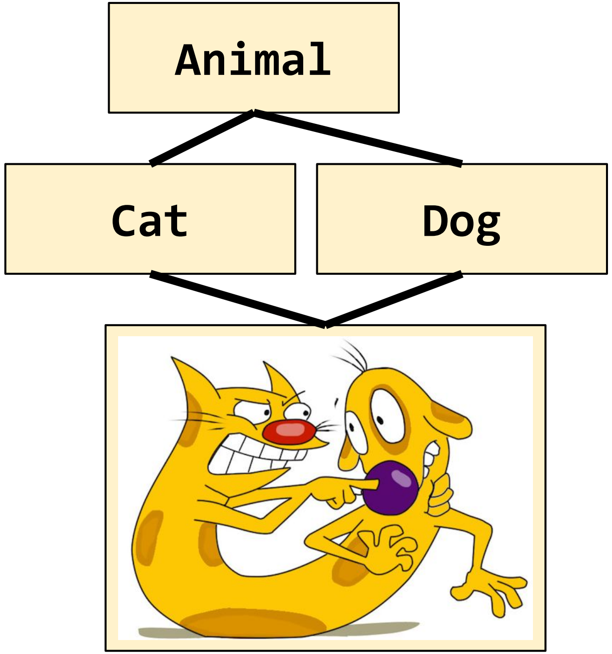
```
Cat IS-AN Animal
```

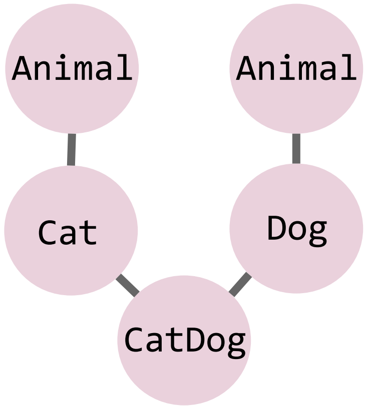


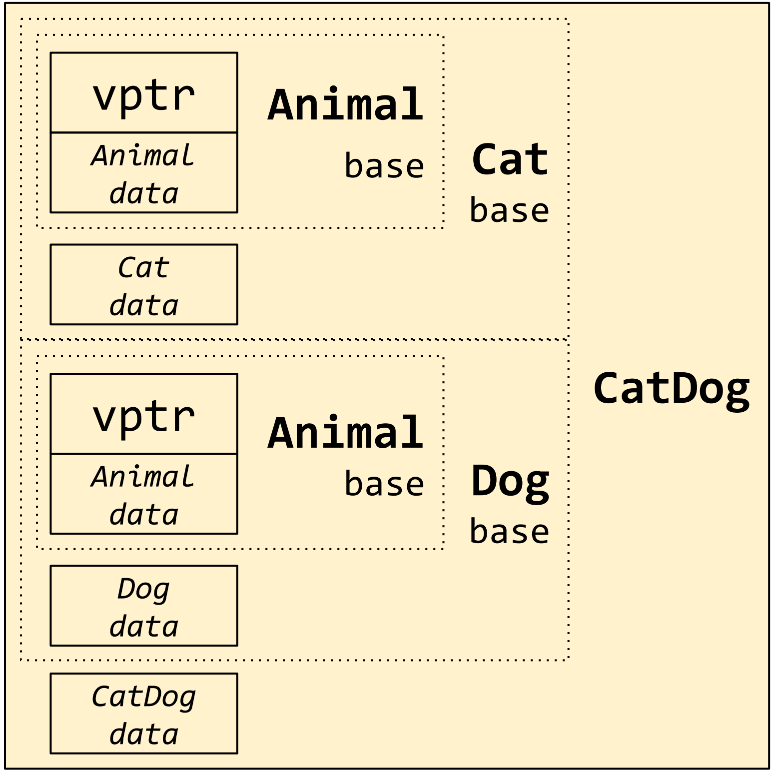


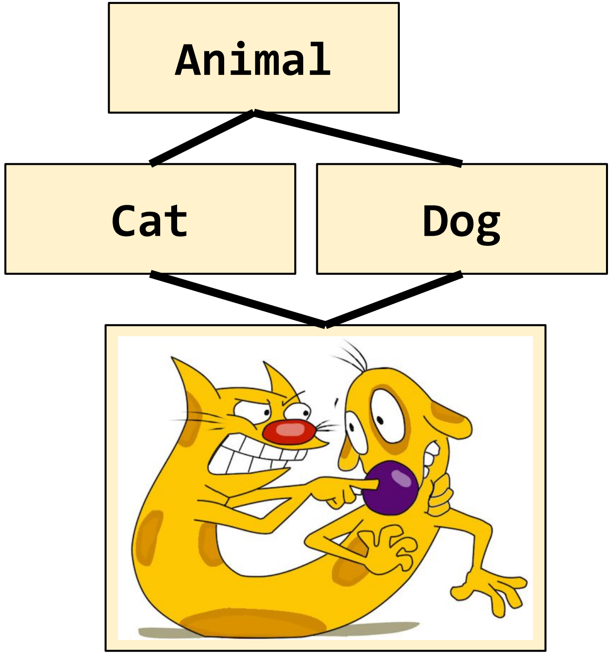
Multiple inheritance

```
class Animal { virtual ~Animal(); };

class Cat : public Animal { };

class Dog : public Animal { };


class CatDog :
  public Cat, public Dog { };
```




Animal
Animal
Cat
Dog
CatDog


vptr
Animal data
Animal base
Cat base
Cat data
vptr
Animal data
Animal base
Dog base
Dog data
CatDog data
CatDog

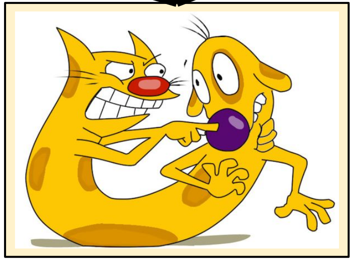

Animal
Cat
Dog

IS-A CatDog an Animal?

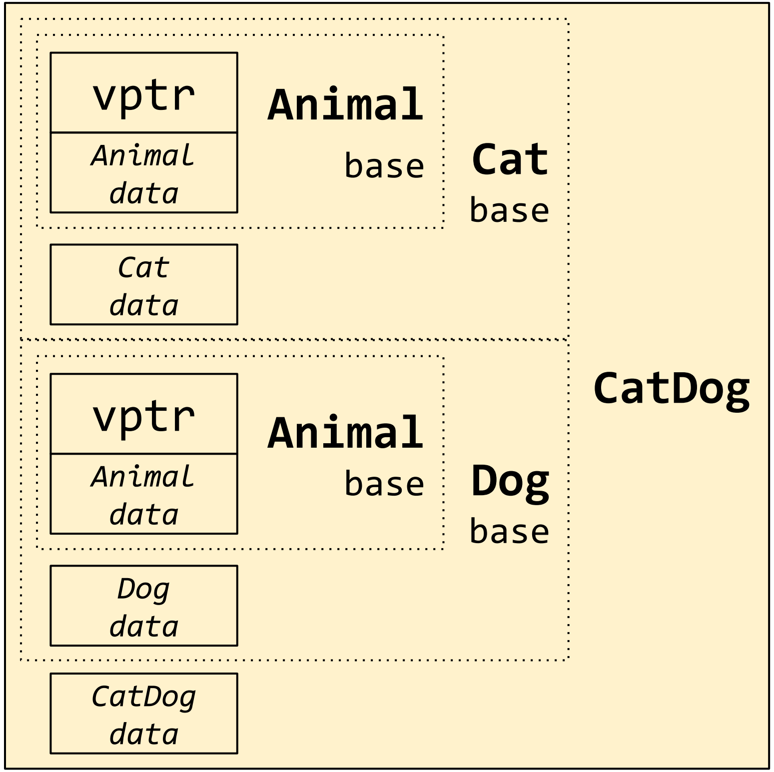
Animal

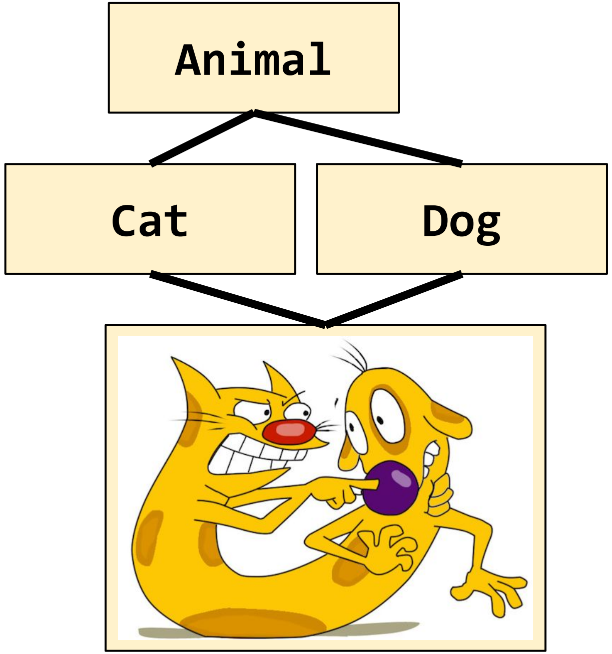
Cat

Dog

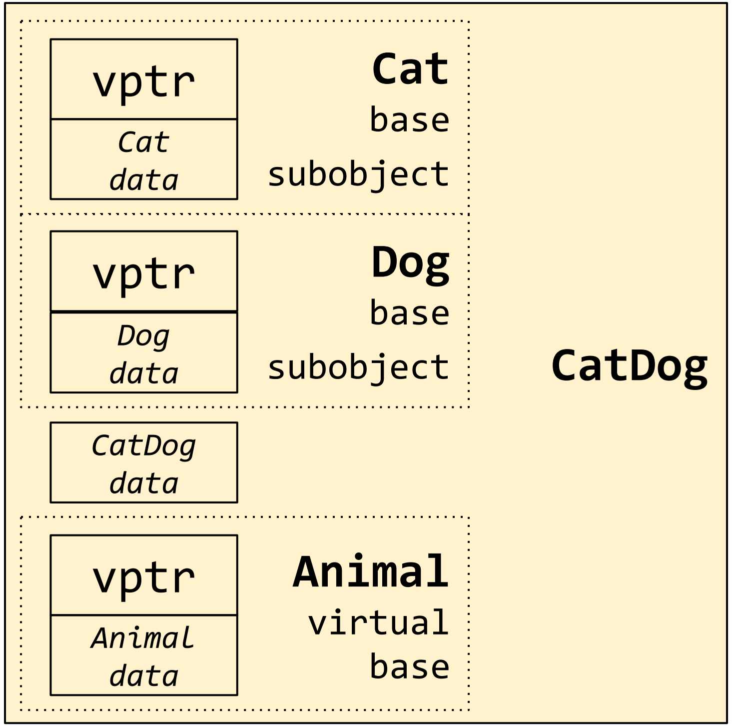


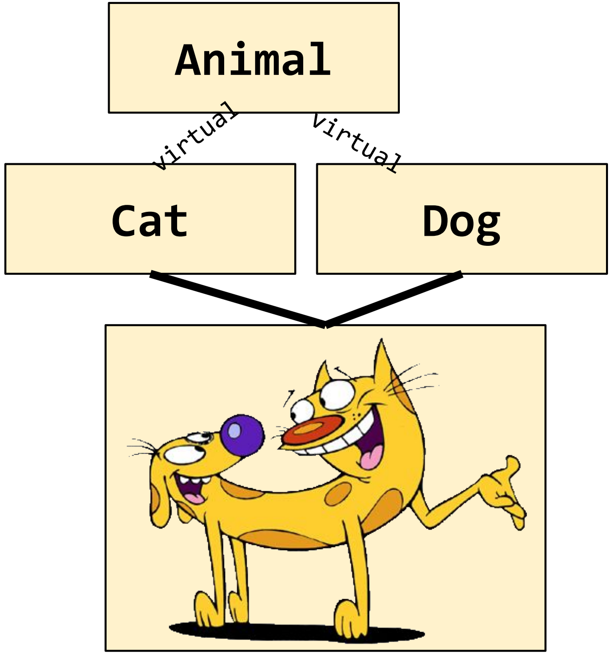
IS-A CatDog an Animal? No. (It’s two Animals.)





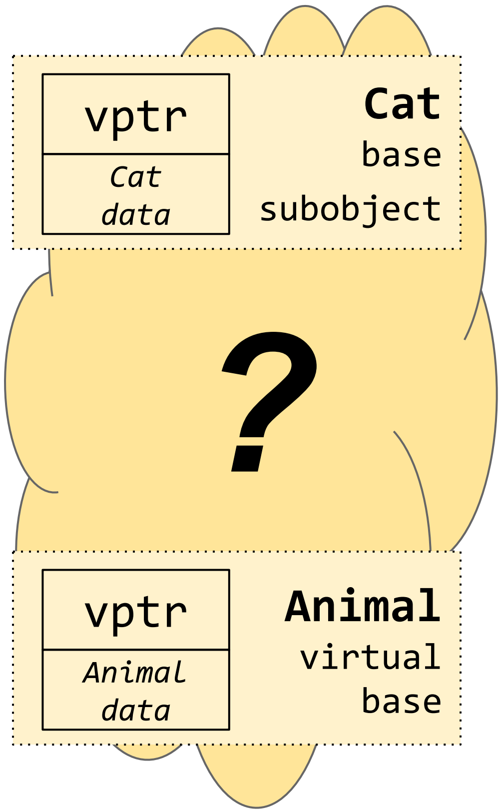
Fix the Dreaded Diamond with *virtual inheritance*

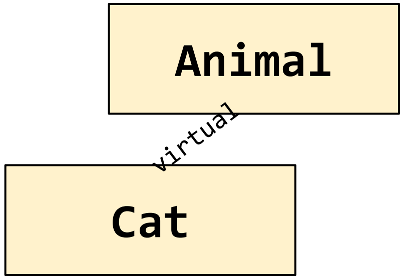




Now IS-A CatDog an Animal? Definitely yes.

But now we’ve changed how a Cat looks!

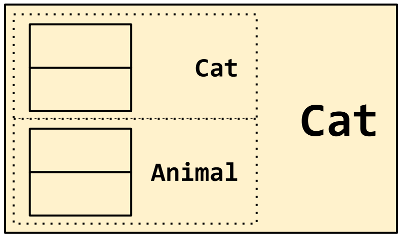




How do we get at the Animal data?

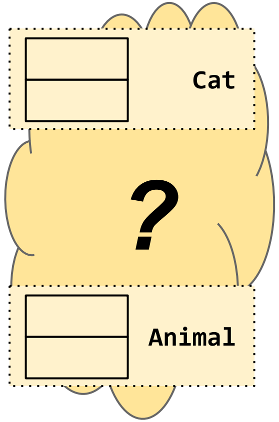
How do we get at the Animal data?

We have to distinguish between objects that are really just `Cat`s and objects that are "`Cat` but maybe more."



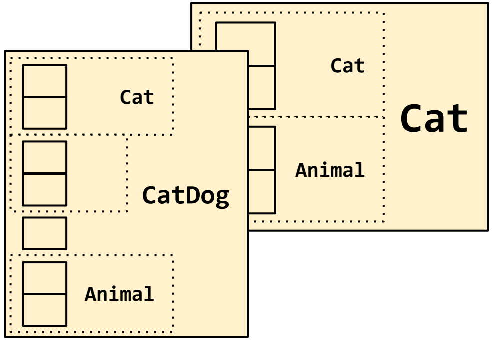
`Cat` objects that are **just `Cat`** have a fixed, concrete layout.

`Cat` objects that are **maybe more** have a fixed layout for all their "base subobject" pieces, but you have to do some work to figure out where the virtual bases are located.



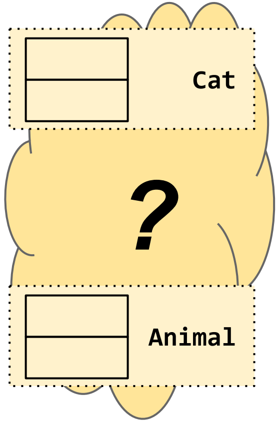
How do we get at the Animal data?

We have to distinguish between objects that are **most derived** and objects whose dynamic type is unknown.



`Cat` objects that are **most derived** have a fixed, concrete layout, dictated by their most derived type.

`Cat` objects of **unknown dynamic type** have a fixed layout for all their “base subobject” pieces, but to access a virtual base, you first have to figure out ***what is the most derived type of this object.*** That tells you its fixed, concrete layout; and *that* tells you where its virtual bases are located.



Vtables are always controlled by the most derived object

The *schema* of a vtable is dictated by the object's static type.

- A vtable "for `Cat` in (foo)" will always have a `speak()` method pointer at offset 0, a destructor at offset 8, and so on.
- A vtable "for `Dog` in (foo)" will always have a destructor at offset 0, and so on.

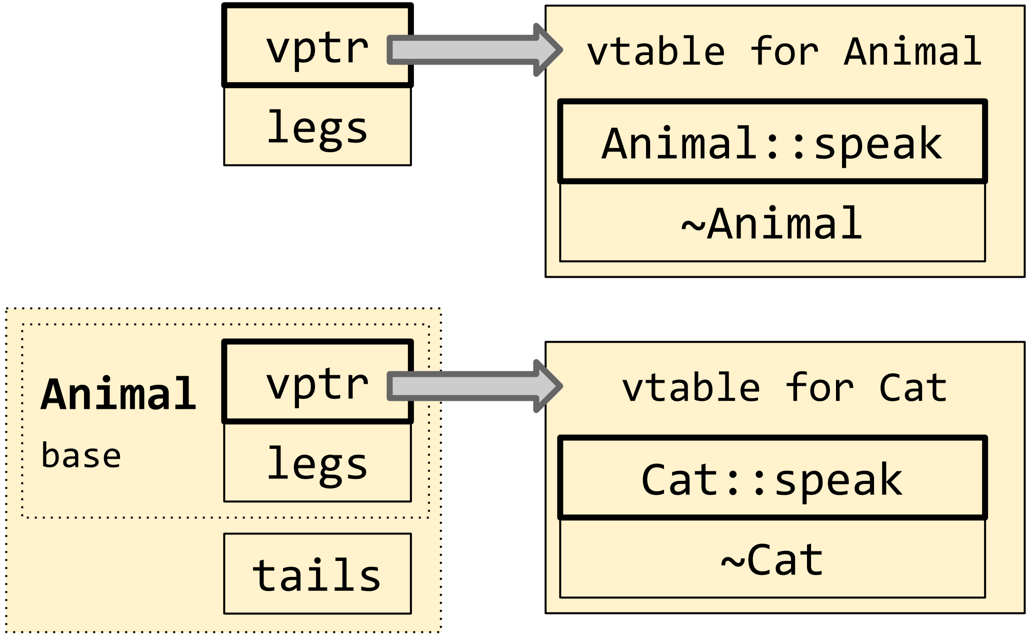
The *data* in a vtable is dictated by the object's dynamic (most-derived) type.

The *presence* of the virtual base `Animal` is dictated statically by the fact that the object **IS-A** `Cat`. The *location* of the virtual base `Animal` is dictated dynamically by the dynamic (most-derived) type.

We can find our virtual base `Animal` by querying our `Cat` object's vtable.

Vtables controlled by most-derived

This was how our memory layout looked on slide 4. Let's update it to reflect the virtual inheritance...



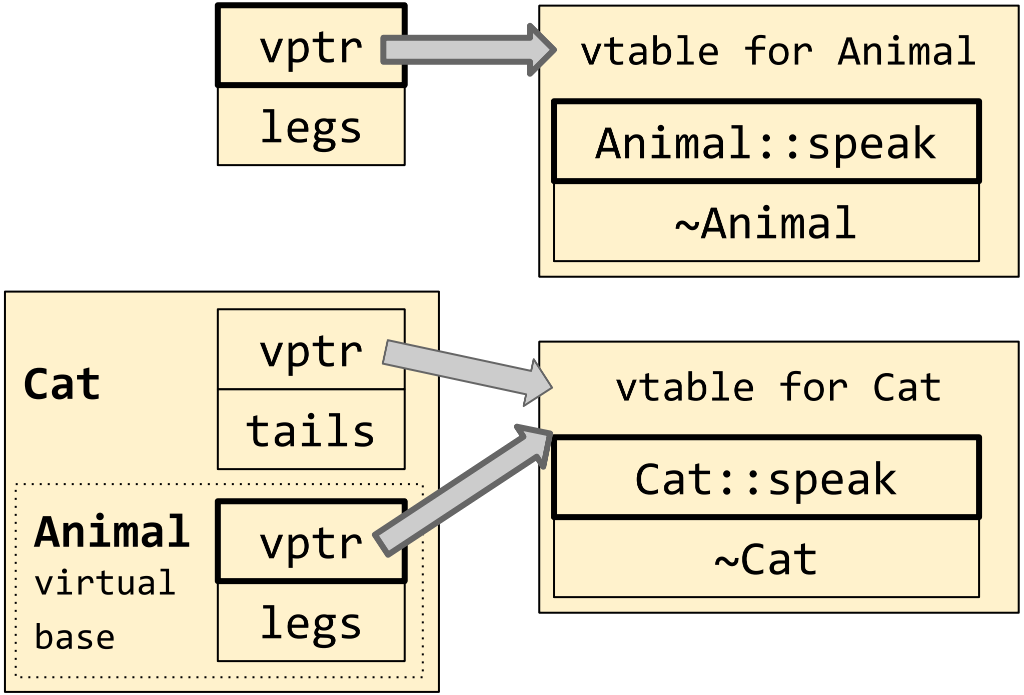
Notice that this is also the vtable for "Animal in Cat," so its schema is a superset of the vtable for "Animal in Animal" above.

Vtables controlled by most-derived

This was how our memory layout looked on slide 4. Let's update it to reflect the virtual inheritance...

Notice that when I have a pointer to any base subobject of `Cat`, as long as that base class has a vptr, the vptr **will** point to a vtable of `Cat`.

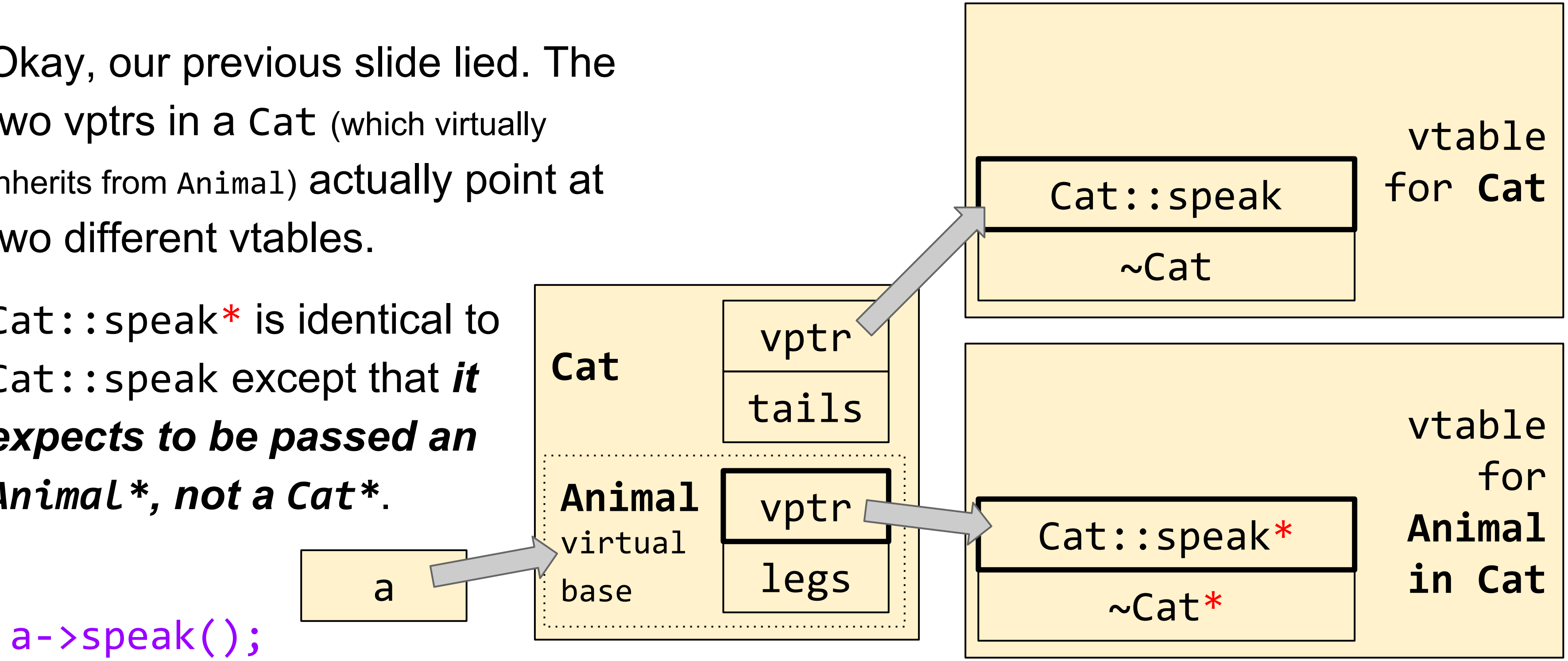
So "what is the most derived type of this object" is answered by looking at the vtable.



Secondary vtables and thunks

Okay, our previous slide lied. The two vptrs in a `Cat` (which virtually inherits from `Animal`) actually point at two different vtables.

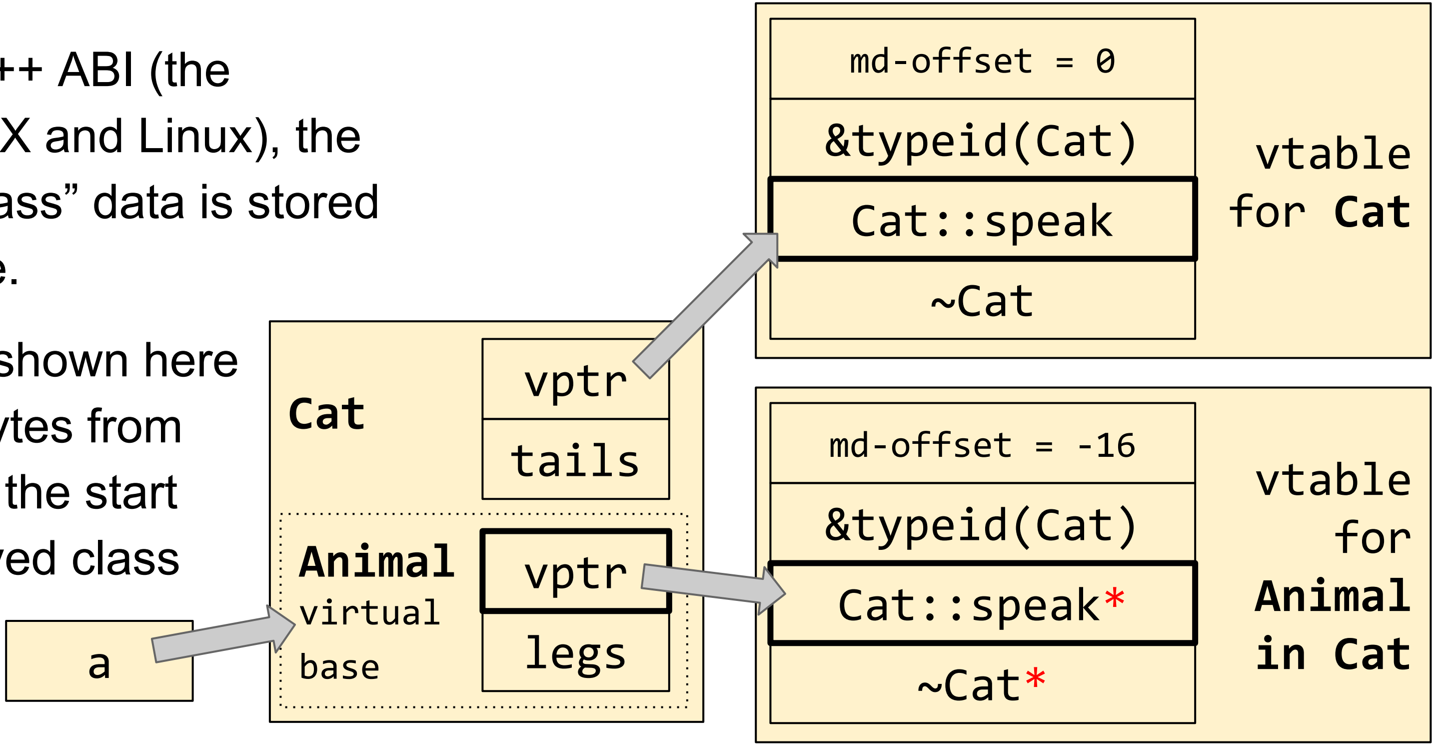
`Cat::speak*` is identical to `Cat::speak` except that ***it expects to be passed an `Animal*`, not a `Cat*`***.



When in doubt: it's in the vtable

In the Itanium C++ ABI (the standard on OS X and Linux), the "most derived class" data is stored *before* the vtable.

The "md-offset" shown here is the offset in bytes from the given vptr to the start of the most derived class — in this case, `Cat`.

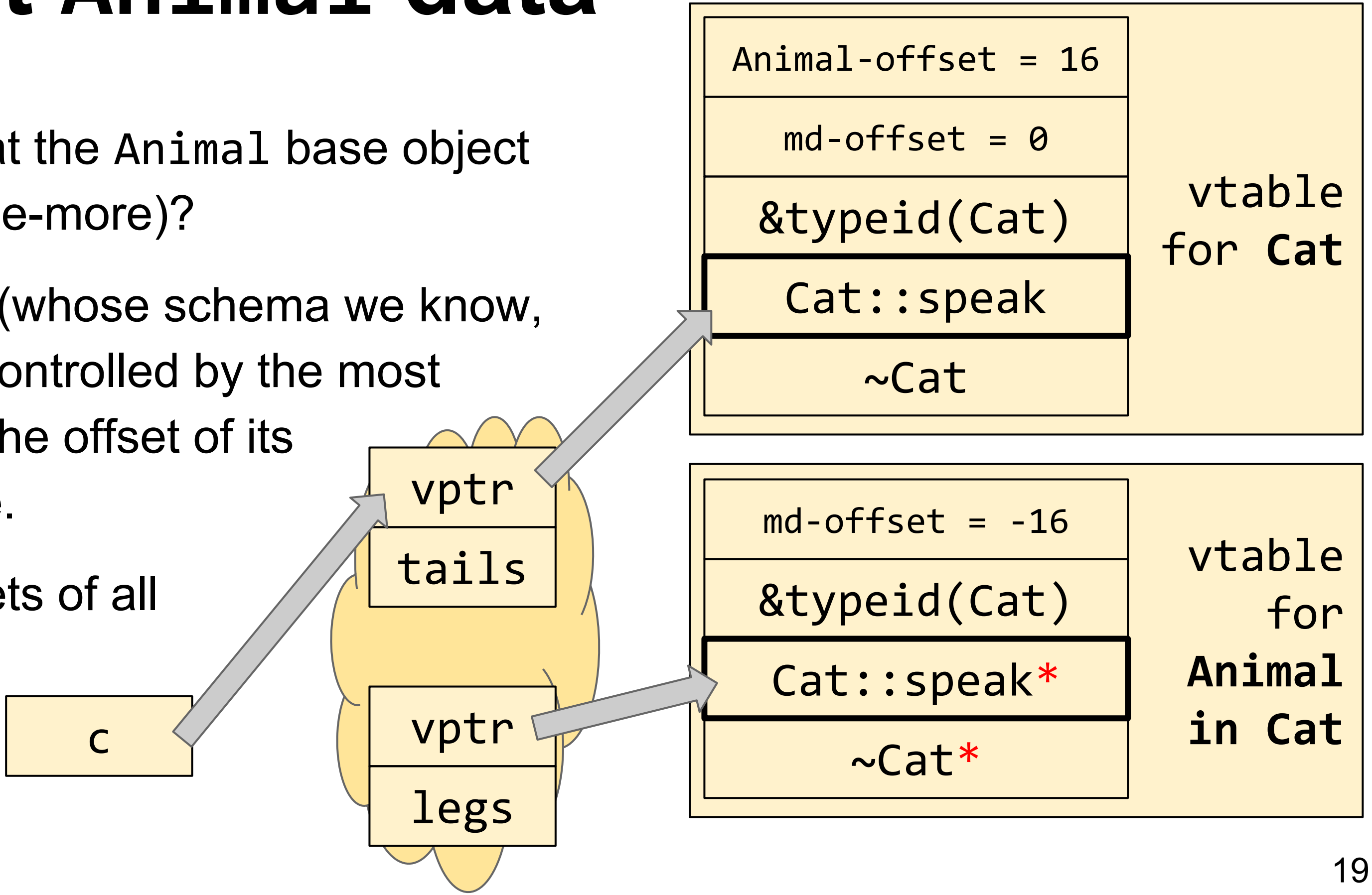


Getting at `Animal` data

So how do we get at the `Animal` base object of a `Cat` (and-maybe-more)?

Ask our `Cat` vtable (whose schema we know, but whose data is controlled by the most derived object) for the offset of its `Animal` virtual base.

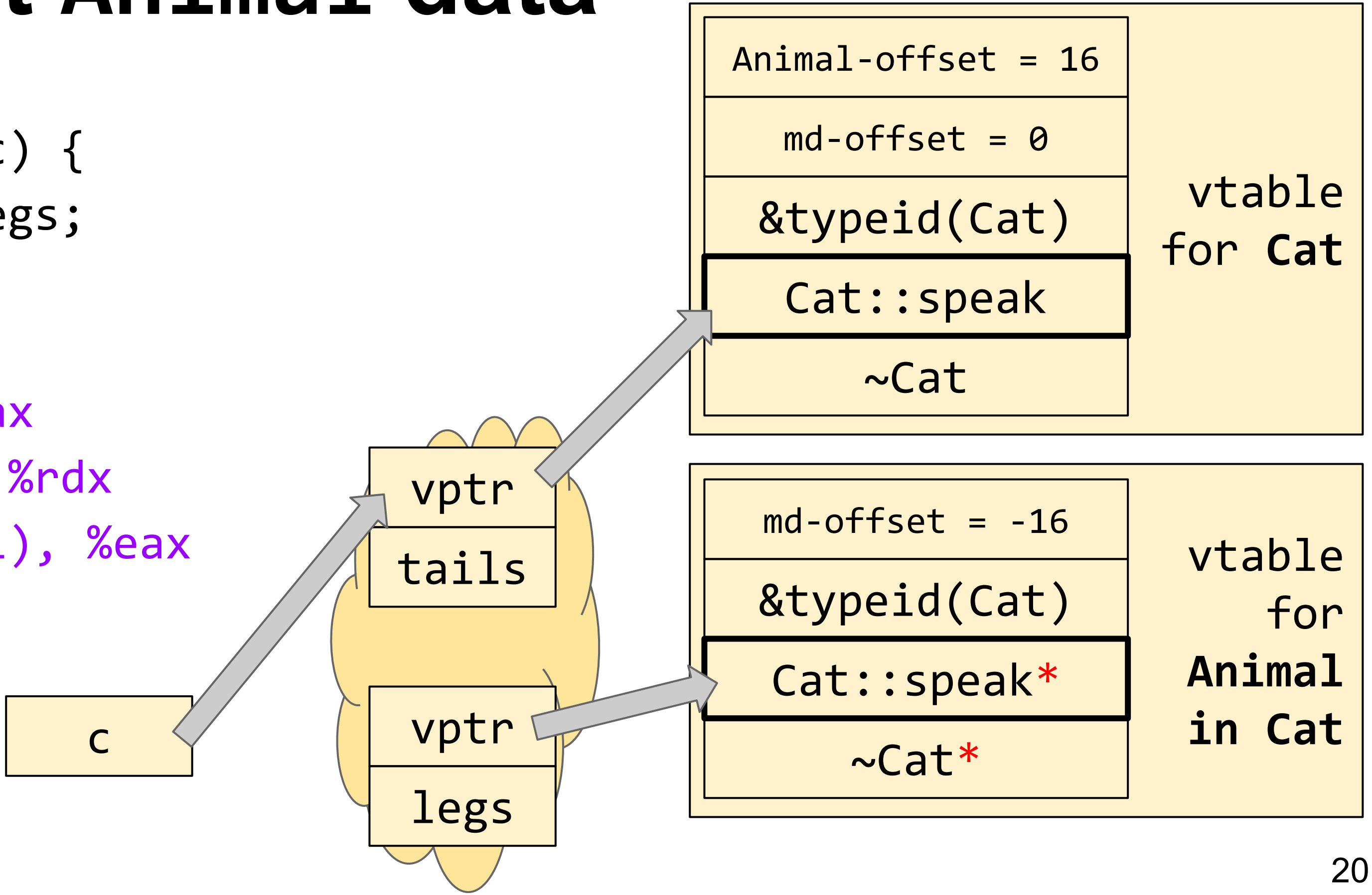
We'll store the offsets of all our virtual bases ahead of the md-offset.



Getting at `Animal` data

```
void test(Cat *c) {
    return c->legs;
}
```

```
movq (%rdi), %rax
movq -24(%rax), %rdx
movl 8(%rdx,%rdi), %eax
```



Vtable layout recap (Itanium ABI)

...

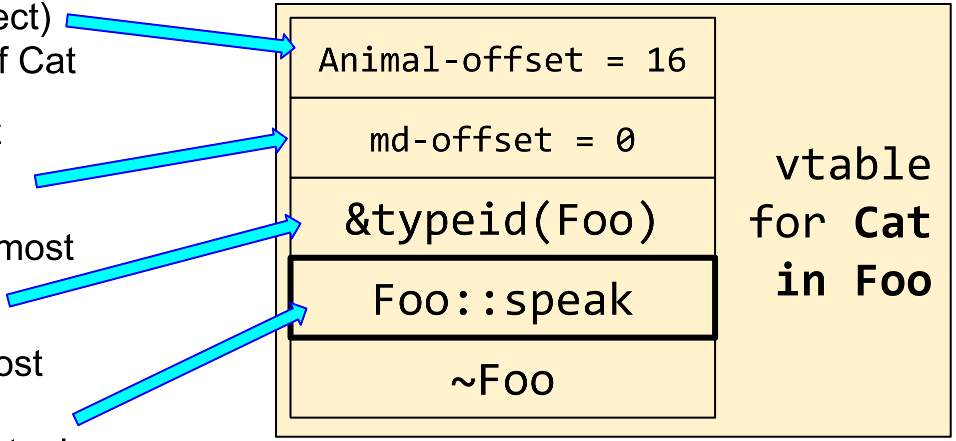
offsets (within the most derived object) to virtual bases of Cat

offset to the most derived object

type_info for the most derived object

pointers to the most derived object’s versions of the virtual methods declared by Cat

...



`dynamic_cast` from scratch

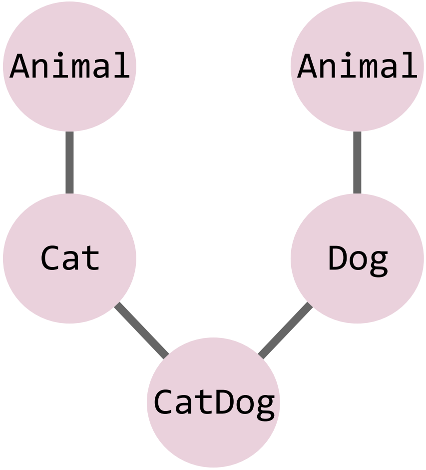
Two kinds of inheritance graphs

How I visualize complex class layout

```
struct Animal { virtual ~Animal(); }
struct Cat : public Animal {};
struct Dog : public Animal {};
struct CatDog : public Cat, Dog {};
```

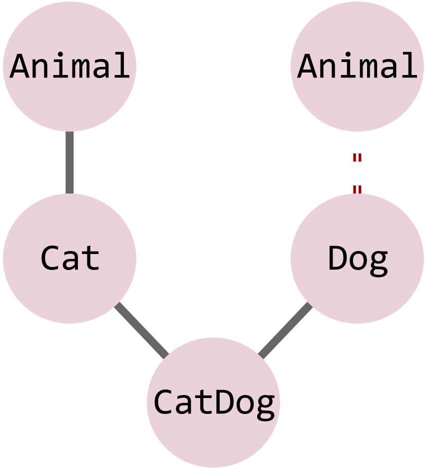
Lines indicate inheritance.

Each “node” indicates a subobject.



How I visualize complex class layout

```
struct Animal { virtual ~Animal(); }
struct Cat : public Animal {};
struct Dog : protected Animal {};
struct CatDog : public Cat, Dog {};
```



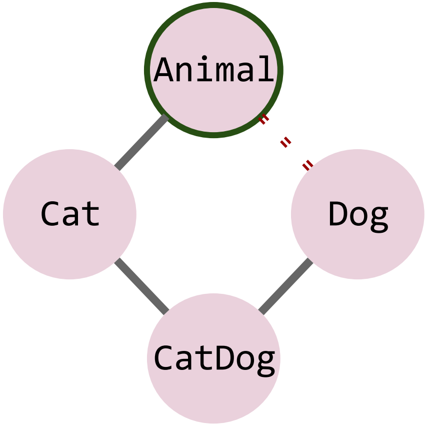
Lines indicate inheritance.

Each “node” indicates a subobject.

Dotted lines indicate non-public inheritance.

How I visualize complex class layout

```
struct Animal { virtual ~Animal(); }
struct Cat : public virtual Animal {};
struct Dog : protected virtual Animal {};
struct CatDog : public Cat, Dog {};
```



Lines indicate inheritance.

Each “node” indicates a subobject.

Dotted lines indicate non-public inheritance.

Heavy circles indicate virtual inheritance.

- Notice that there is only ever *one* virtual subobject with a given name.

How I visualize complex class layout

Lines indicate inheritance.

Each “node” indicates a subobject.

Dotted lines indicate non-public inheritance.

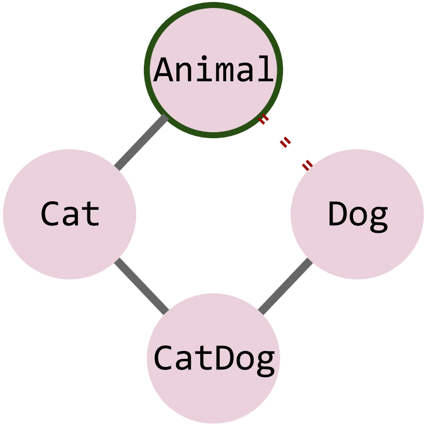
Heavy circles indicate virtual inheritance.

- Notice that there is only ever *one* virtual subobject with a given name.

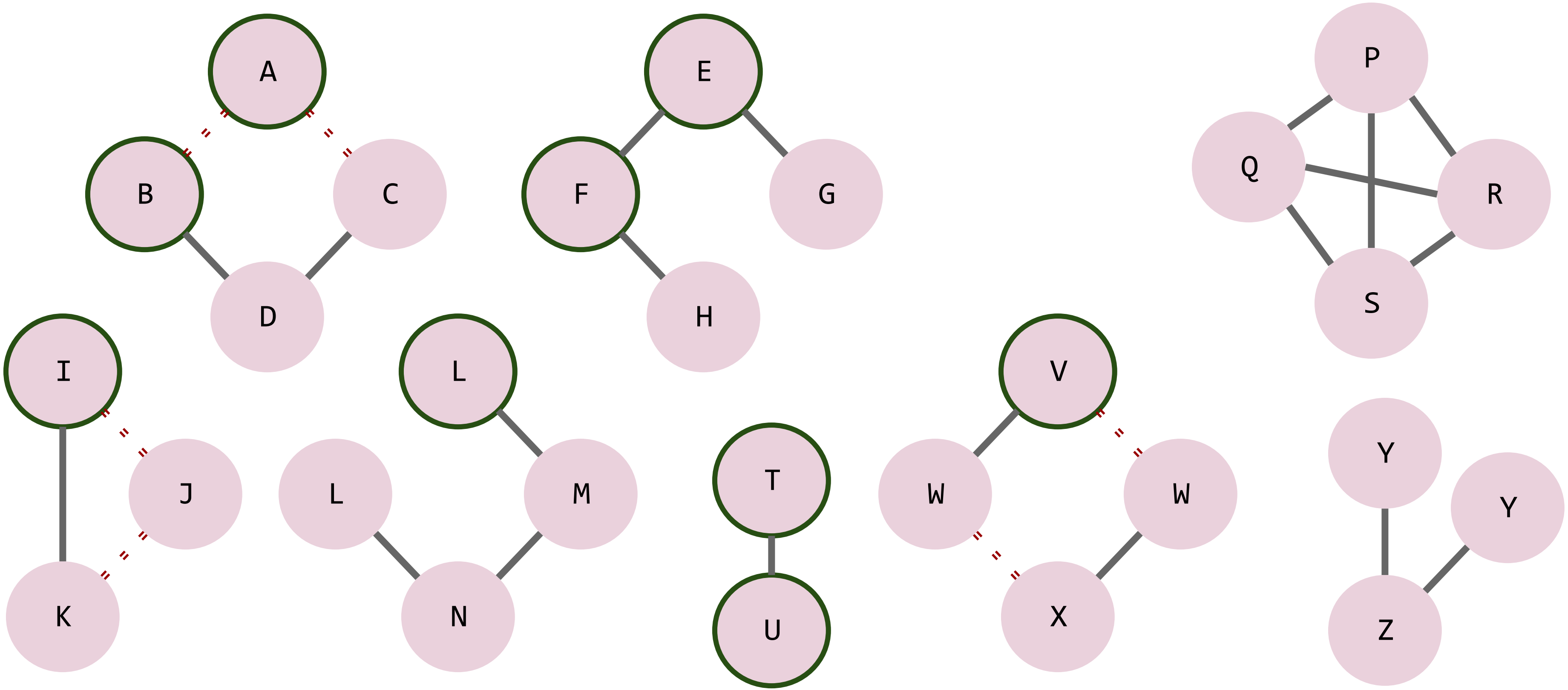
The root of the graph is the *most derived object*.

- Notice that there is always a single root.

We don’t have a great way to indicate inheritance order,
but that’s okay, because inheritance order doesn’t matter to
(a correct implementation of) `dynamic_cast`.

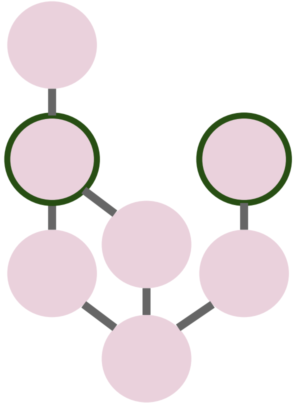


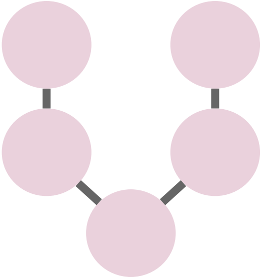
Which of these graphs are possible?

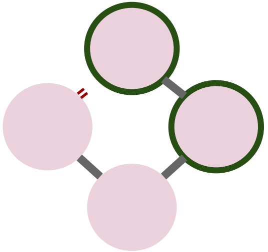


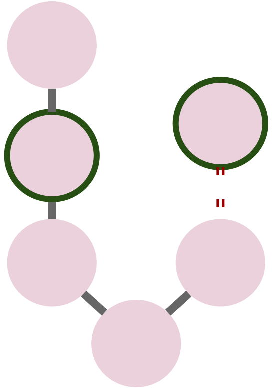
Match these graphs to their code

```
struct Animal;
struct Cat : Animal {};
struct Dog : Animal {};
struct Sponge : protected virtual Animal {};
struct LeftCat : virtual Cat {};
struct RightCat : virtual Cat {};
struct Flea : virtual Animal {};

struct CatDog : Cat, Dog {};
struct SiameseCat : LeftCat, RightCat, Flea {};
struct Bath : LeftCat, Sponge {};
struct Nemo : Sponge, virtual Flea {};
```



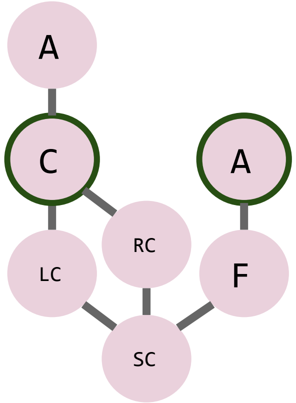


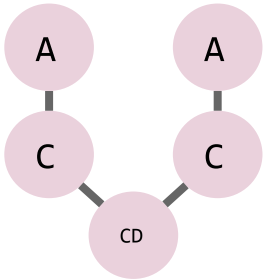


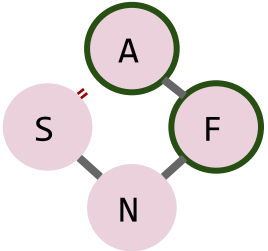


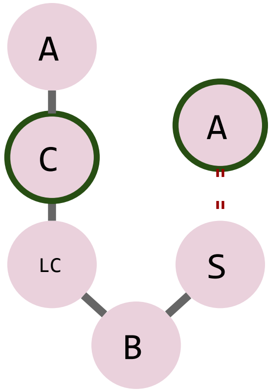
Match these graphs to their code

```
struct Animal;
struct Cat : Animal {};
struct Dog : Animal {};
struct Sponge : protected virtual Animal {};
struct LeftCat : virtual Cat {};
struct RightCat : virtual Cat {};
struct Flea : virtual Animal {};

struct CatDog : Cat, Dog {};
struct SiameseCat : LeftCat, RightCat, Flea {};
struct Bath : LeftCat, Sponge {};
struct Nemo : Sponge, virtual Flea {};
```







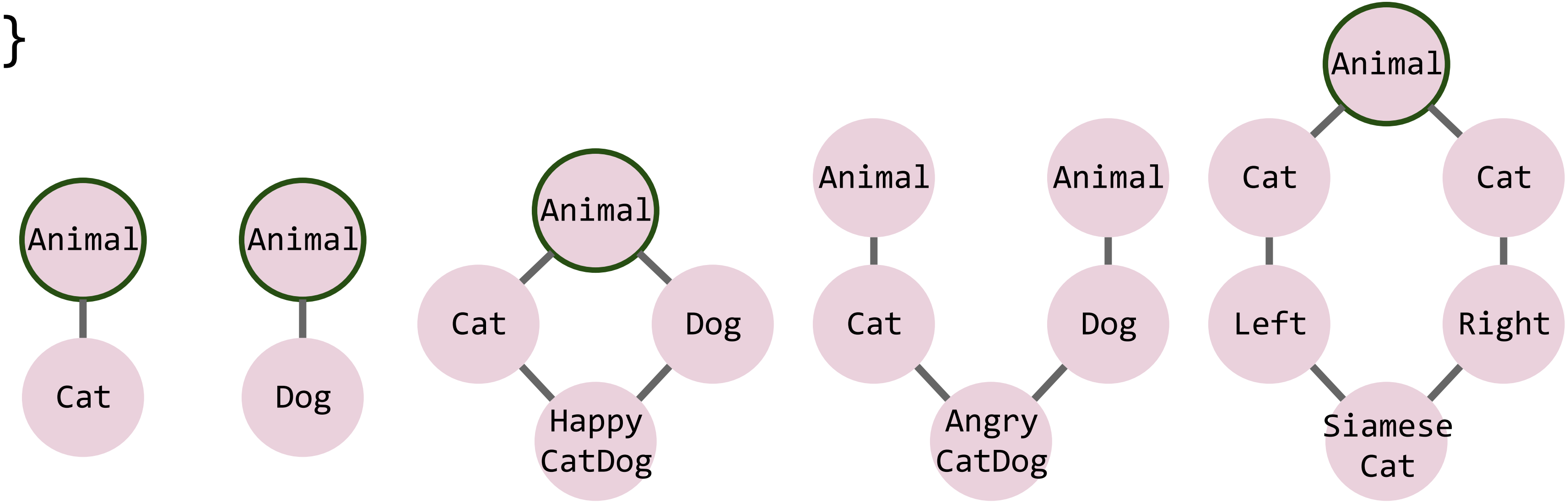


Outline

- Part I: Recap polymorphism
  - Inheritance, vtables, and the Dreaded Diamond [3–10]
  - Accessing members of virtual bases [11–21]
  - How to visualize complex class layouts [22–29]
- **Part II: `dynamic_cast`**
  - What should `dynamic_cast` do? [31–54]
  - Okay, how do we implement that? [55–60]
  - Benchmark numbers [61–64]

What should dynamic_cast do?

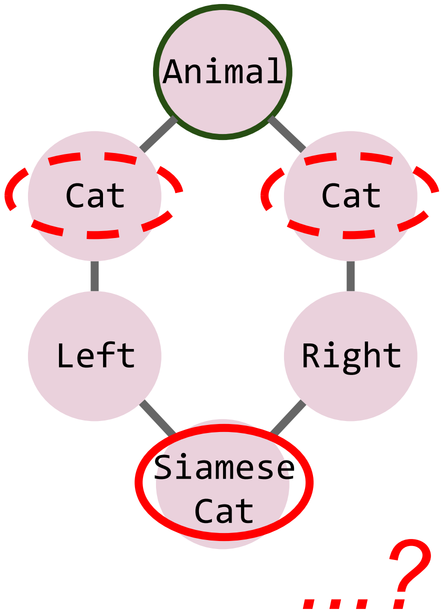
```
auto test(Animal *animal) {
    return dynamic_cast<Cat*>(animal);
}
```



What should catch do?

Ambiguous bases are basically invisible to RTTI.
This also applies during catch-handler matching.

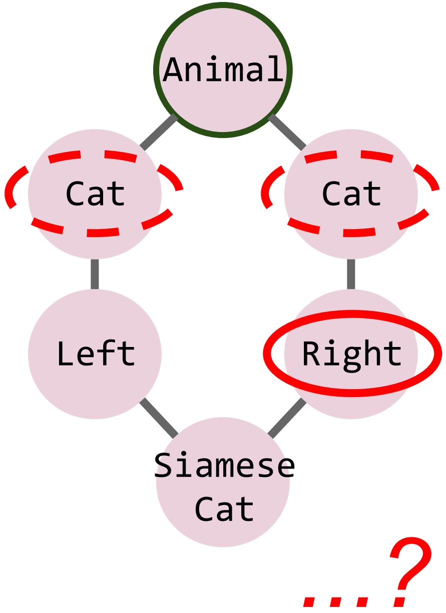
```
int main() {
    try {
        throw SiameseCat();
    } catch (const Cat&) {
        puts("SiameseCat IS-NOT-A Cat...");
        // it's two cats!
    } catch (const Animal&) {
        puts("...but SiameseCat IS-AN Animal!");
    }
}
```



What should this cast do?

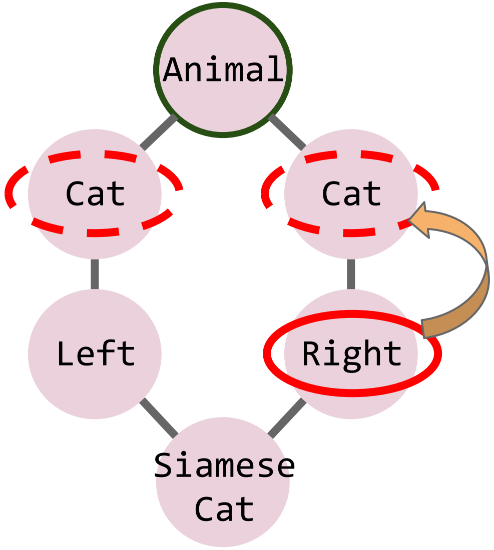
This raises one more question: What about this case?

```
Cat *derived_to_sibling_or_to_base(RightCat *a) {
    return dynamic_cast<Cat *>(a);
}
```



What should this cast do?

```
Cat *derived_to_sibling_or_to_base(RightCat *a)
{
    return dynamic_cast<Cat *>(a);
}
```



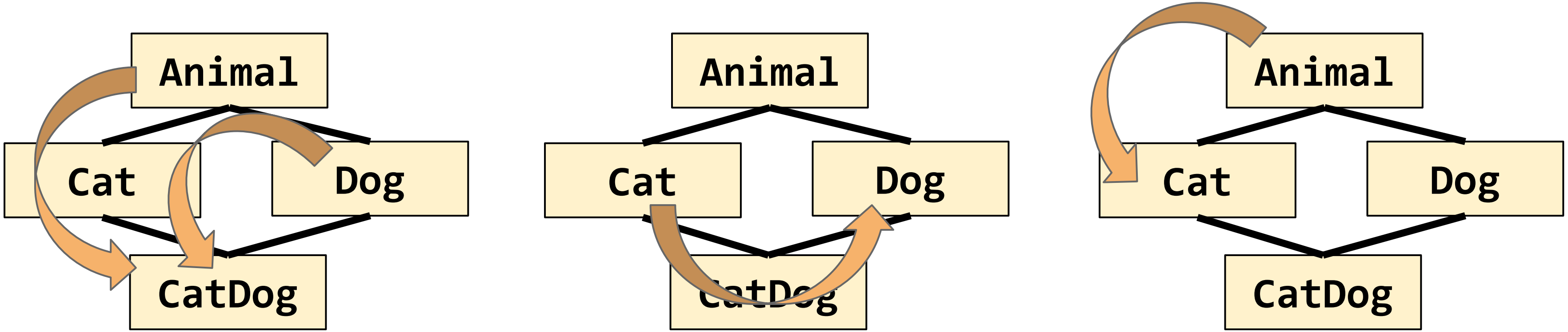
Trick question, sort of. This is not a “truly dynamic cast.” Because `Cat` is a known base class of `RightCat`, this cast is 100% equivalent to a `static_cast`. The compiler verifies that `Cat` is an accessible, unambiguous base class of `RightCat` and then generates the optimal code.

If `Cat` is an *inaccessible* base class of `RightCat`, the dynamic_cast is ill-formed and you get a compile-time error.

dynamic_cast, take 1

Per the Itanium spec, there are just three “truly dynamic casts” —

- `dynamic_cast<void*>` to the most-derived class
- `dynamic_cast` across the hierarchy, to a sibling base
- `dynamic_cast` from base to derived

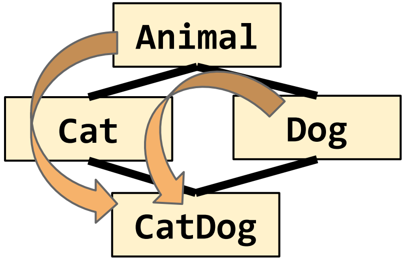


TO MDO **TO SIBLING** **TO DERIVED**

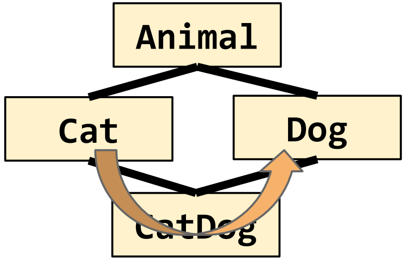
The fourth “truly dynamic cast”

We need this one when deciding whether to stop stack-unwinding at a given `catch` clause. Let’s call it `castToBase`.

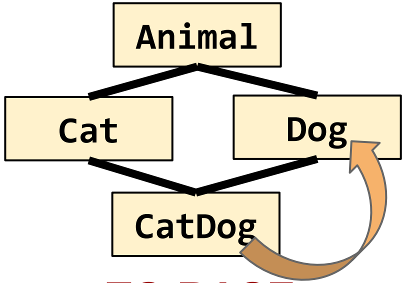
- `dynamic_cast<void*>` to the most-derived class
- `dynamic_cast` across the hierarchy, to a sibling base
- *The dynamic MDO-to-public-base conversion implied by `catch(Base&)`*
- `dynamic_cast` from base to derived



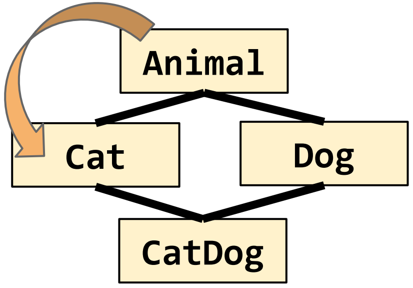
TO MDO



TO SIBLING



TO BASE

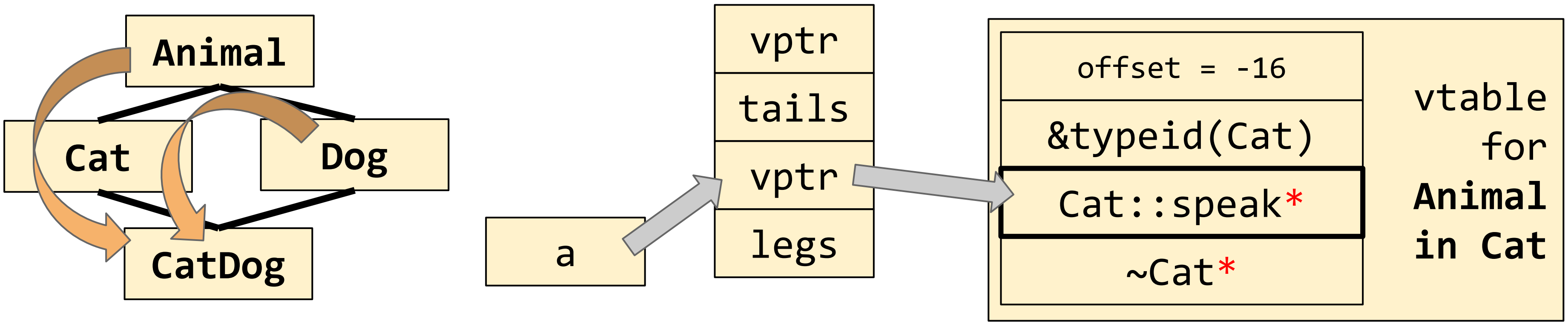


TO DERIVED

`dynamic_cast` to most-derived class

`dynamic_cast<void*>(p)` is literally just "load the offset-to-most-derived word located at `vptr[-2]` and add it to the `this` pointer."

Every major compiler will generate optimal inline code for this.

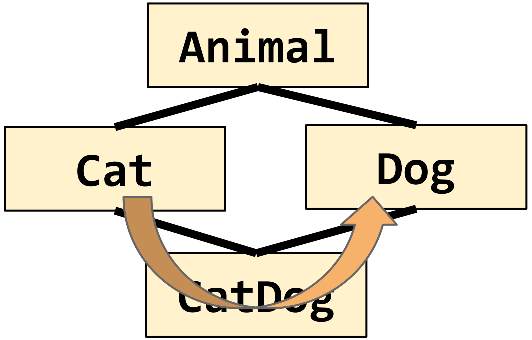


DONE. SO EASY

`dynamic_cast` to sibling base

In this case we have a `Cat&` and we're trying to convert it to a `Dog&`.
The compiler already knows, statically, that `Cat` and `Dog` are "unrelated."
That is, not all `Cat`s are `Dog`s and not all `Dog`s are `Cat`s.
So the *only* way for this `dynamic_cast` to succeed is if somebody has made a `CatDog`.

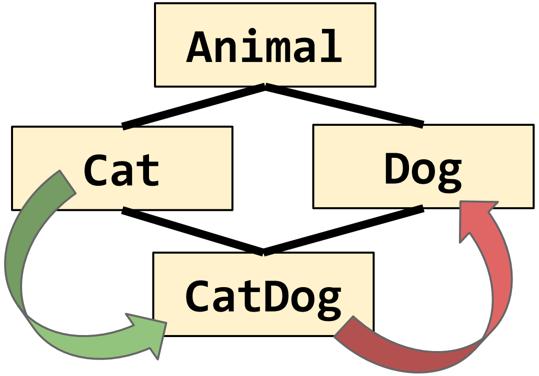
Clang optimizes iff `Cat` is final.
ICC, GCC, and MSVC do not optimize.



TO SIBLING

`dynamic_cast` to sibling base

- First, get a pointer to the most-derived class.
- If that most-derived class is still `Cat`, we can fail.
- Otherwise, "`castToBase`"!

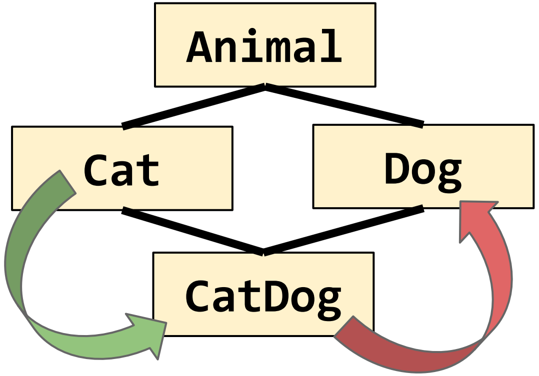


CAST TO MDO + CAST TO BASE

`dynamic_cast` MDO to base

We can imagine putting a compiler-generated function at a well-known offset in every vtable (or putting such a function pointer into every `std::type_info`):

```
void *castToBase(void *mdo, const type_info& to);
```



CAST TO MDO + CAST TO BASE

`dynamic_cast` MDO to base

```
void *Animal_castToBase(char *animal, const type_info& to) {
    return nullptr;
}
void *Cat_castToBase(char *cat, const type_info& to) {
    if (to == typeid(Animal)) return cat + 16;
    return nullptr;
}
void *HappyCatDog_castToBase(char *catdog, const type_info& to) {
    if (to == typeid(Cat)) return catdog + 0;
    if (to == typeid(Dog)) return catdog + 16;
    if (to == typeid(Animal)) return catdog + 32;
    return nullptr;
}
```

`dynamic_cast` MDO to base

```
void *Animal_castToBase(char *animal, const type_info& to) {
    return nullptr;
}
void *Cat_castToBase(char *cat, const type_info& to) {
    if (to == typeid(Animal)) return cat + 0;
    return nullptr;
}
void *AngryCatDog_castToBase(char *catdog, const type_info& to) {
    if (to == typeid(Cat)) return catdog + 0;
    if (to == typeid(Dog)) return catdog + 24;
    return nullptr;
}
```

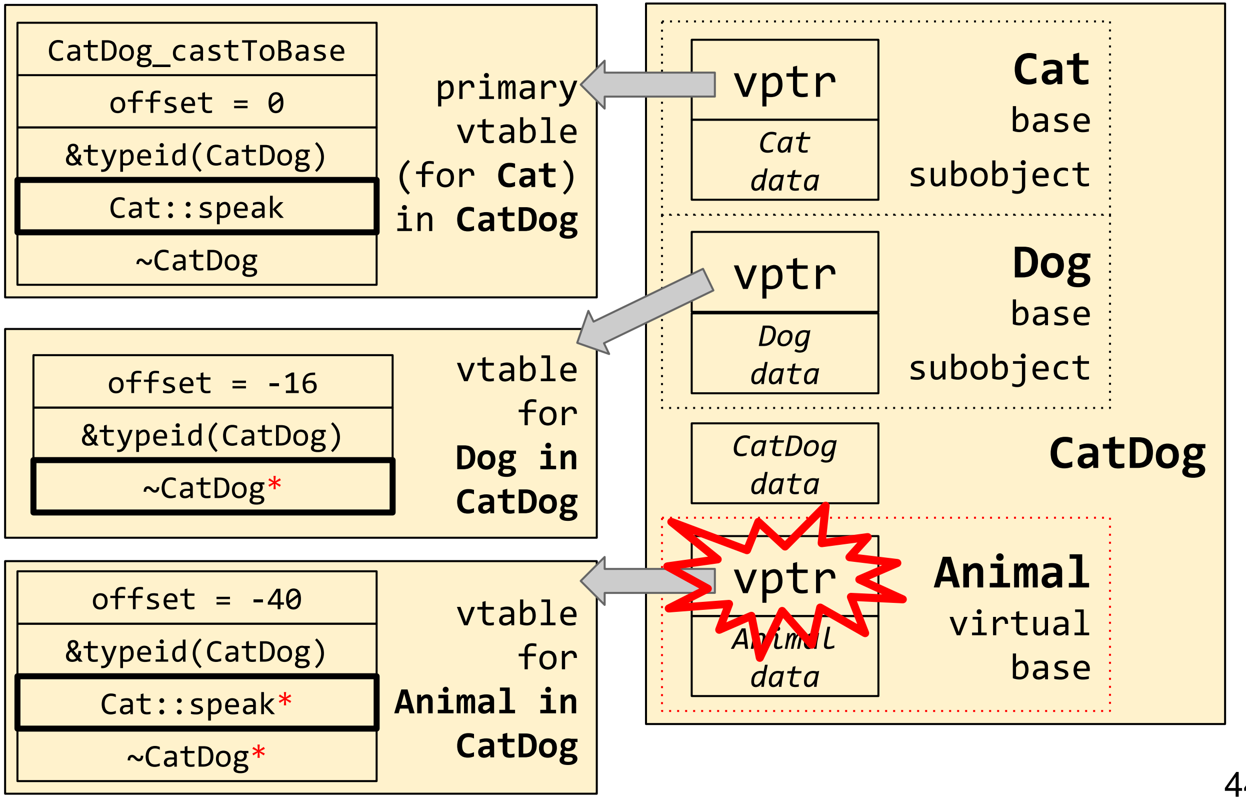
Let’s do an example.

Casting `Animal` to `Dog` (in a `CatDog`)

dynamic_cast<Dog*>(Animal*)

```
Dog *test(Animal *p) {
    return dynamic_cast<
         Dog*>(p);
}
```

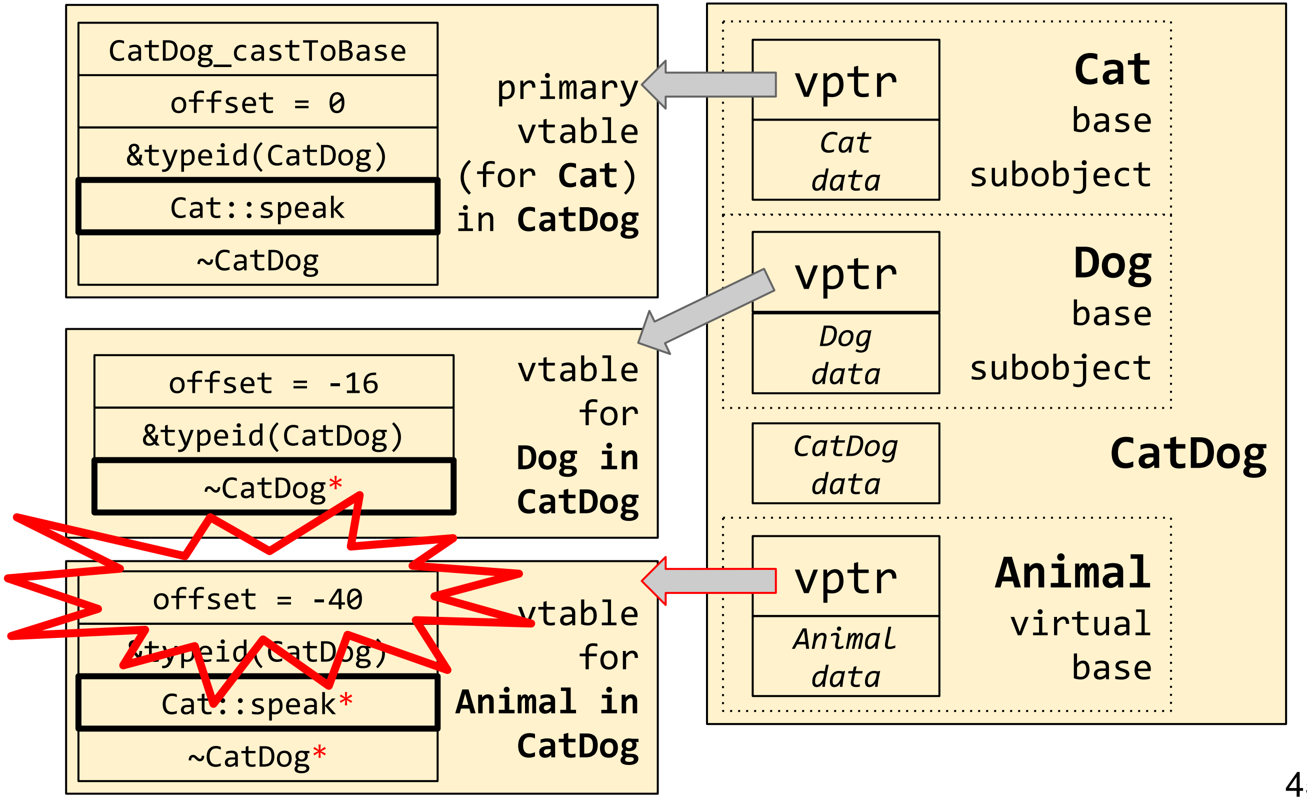
1. Get the vptr from p.



dynamic_cast<Dog*>(Animal*)

```
Dog *test(Animal *p) {
   return dynamic_cast<
      Dog*>(p);
}
```

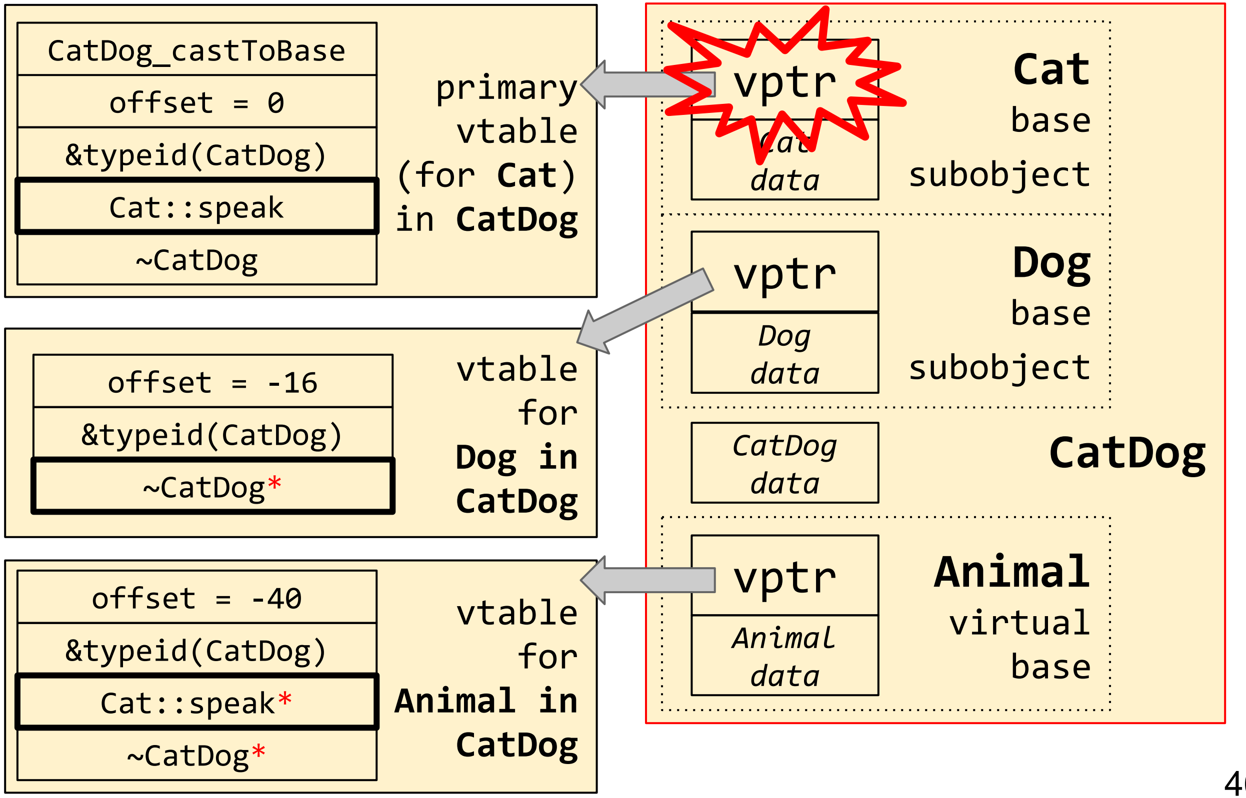
1. Get the vptr from p.
2. Load the offset-to-most-derived from `vptr[-2]` and adjust the `this` pointer.



dynamic_cast<Dog*>(Animal*)

```
Dog *test(Animal *p) {
   return dynamic_cast<
       Dog*>(p);
}
```

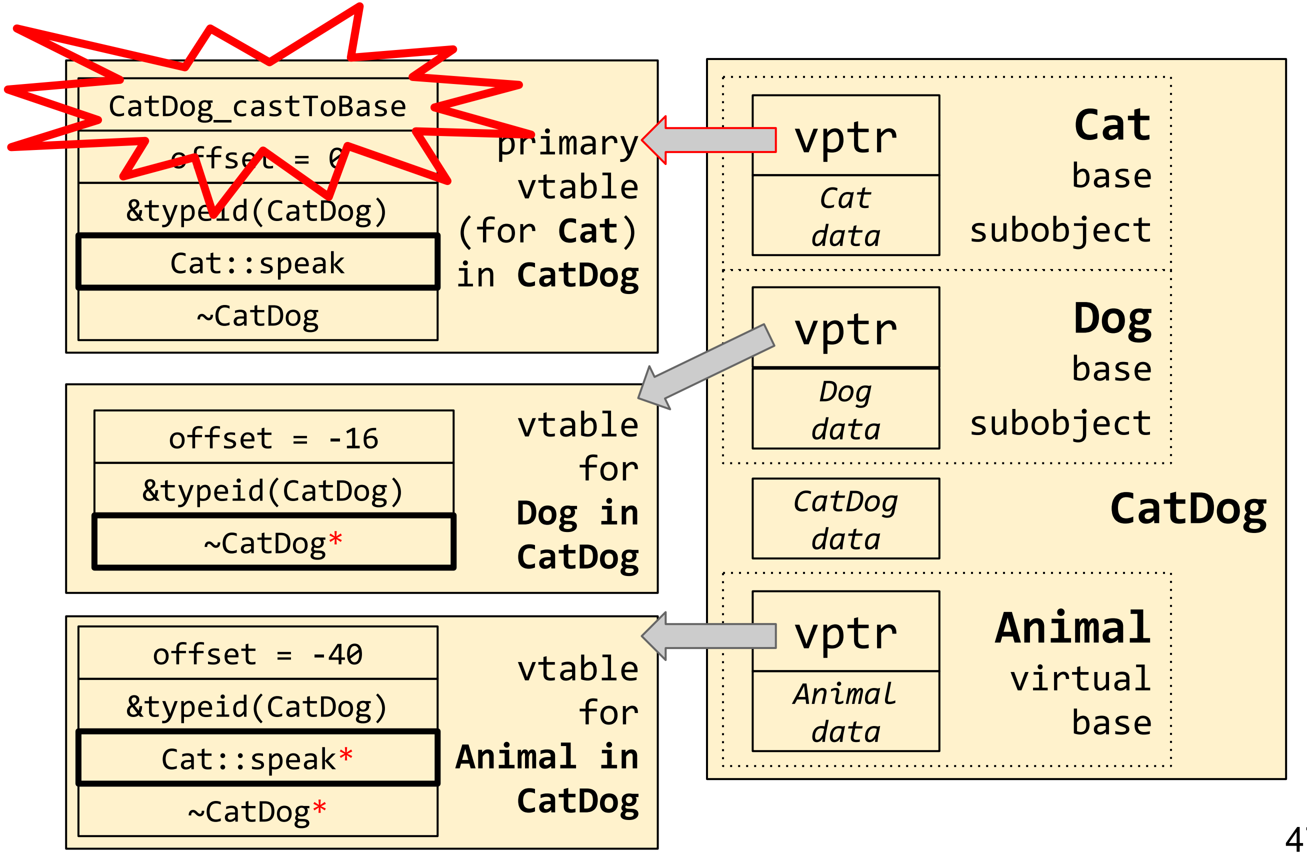
1. Get the vptr from p.
2. Load the offset-to-most-derived from `vptr[-2]` and adjust the `this` pointer.
3. Load the new vptr.



dynamic_cast<Dog*>(Animal*)

```
Dog *test(Animal *p) {
   return dynamic_cast<
       Dog*>(p);
}
```

1. Get the vptr from p.
2. Load the offset-to-most-derived from `vptr[-2]` and adjust the `this` pointer.
3. Load the new vptr.
4. Load the address of `castToBase` from `vptr[-3]`.
5. Call that function with `adjusted_this`, `typeid(Dog)`.



Wind up in `CatDog_castToBase`

```
void *CatDog_castToBase(void *p, const type_info& to) {
    if (to == typeid(Cat)) return p + 0;
    if (to == typeid(Dog)) return p + 16;
    if (to == typeid(Animal)) return p + 32;
    return nullptr;
}
```

This concludes the cases “truly dynamic cast to public base” and “cast between unrelated sibling bases.”

But actually this is NOT how either Itanium or MSVC do it! They encode the entire class hierarchy in data and run expensive code to parse that class hierarchy.
Why? Historical reasons, I guess.

“Accessible” versus “public”

```
struct Animal { virtual ~Animal(); };

struct Sponge : protected Animal {
    auto accessible(Sponge *s) {
        return static_cast<Animal*>(s);  // okay
    }
};
auto inaccessible(Sponge *s) {
    return static_cast<Animal*>(s);  // error
}
```

Either `static_cast` or `dynamic_cast` from derived to base requires that the target base be unambiguous and *accessible* in the current lexical scope.

“Accessible” versus “public”

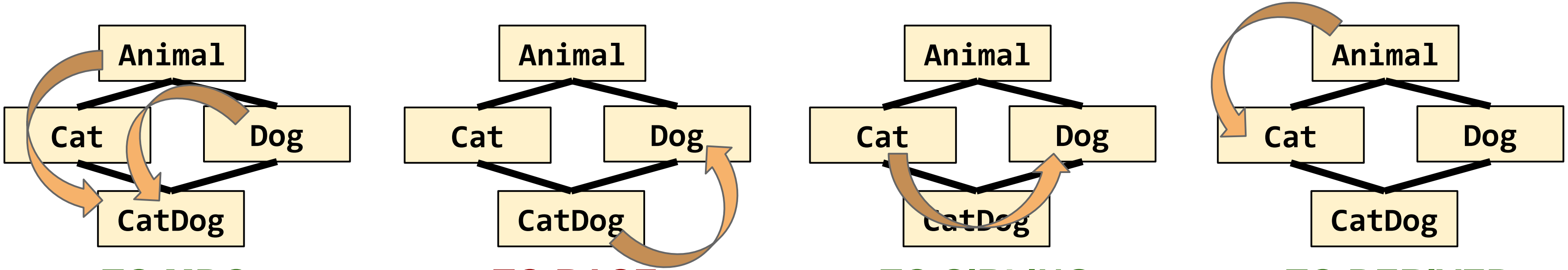
```
struct Animal { virtual ~Animal(); };

struct Sponge : protected Animal {
    void accessible(Sponge *s) {
        try { throw s; } catch (Animal *a) {}  // doesn't catch
    }
};
auto inaccessible(Sponge *s) {
    try { throw s; } catch (Animal *a) {}  // doesn't catch
}
```

`castToBase` (used by `catch` and `dynamic_cast` to sibling base) requires that the target base be unambiguous and *public* — there’s no such thing as lexical scope at runtime!

`dynamic_cast` from parent to child

- `dynamic_cast<void*>` to the most-derived class
- The dynamic derived-to-base conversion implied by `catch(Base&)`
- `dynamic_cast` across the hierarchy, to a sibling base
- `dynamic_cast` *from base to derived*

Casting from base to derived cannot use the same “cast to most-derived, then `castToBase`” trick that we used for sibling bases.

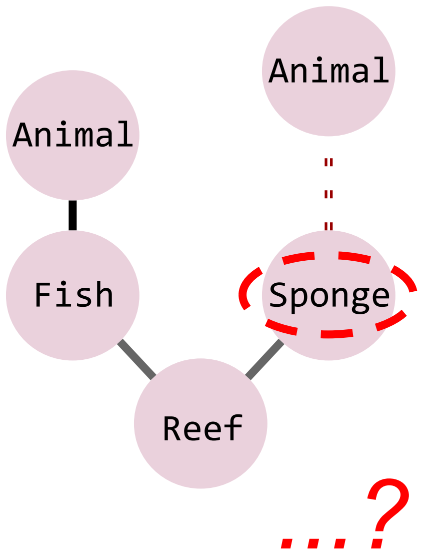


TO MDO TO BASE TO SIBLING TO DERIVED

`dynamic_cast` from parent to child

Casting from base to derived cannot use the same "cast to most-derived, then `castToBase`" trick that we used for sibling bases. Consider:

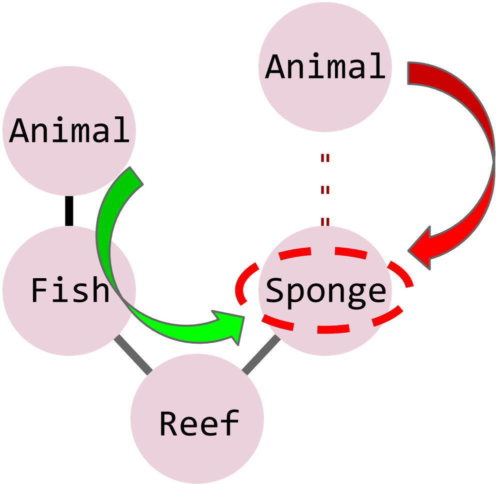
```
struct Sponge : protected Animal {};
struct Fish : public Animal {};
struct Reef : Fish, Sponge {};

Sponge *foo(Animal *animal) {
    return dynamic_cast<Sponge*>(animal);
}
```



`dynamic_cast` from parent to child

Casting from base to derived cannot use the same "cast to most-derived, then `castToBase`" trick that we used for sibling bases. Consider:

```
struct Sponge : protected Animal {};
struct Fish : public Animal {};
struct Reef : Fish, Sponge {};

Sponge *foo(Animal *animal) {
    return dynamic_cast<Sponge*>(animal);
}
```



Our `Sponge` **IS-AN** `Animal` in certain lexical scopes, but at runtime there are no lexical scopes. So it's *not* an `Animal`, and so, likewise, that `Animal` subobject doesn't know it's part of a `Sponge`.

`dynamic_cast` from parent to child

Start by finding the most derived object. (Get its RTTI from the primary vtable.) Ask it two questions:

- Your `Animal` subobject at offset *x* — is it a public base of a `Sponge` subobject? (If so, success: return the address of that `Sponge` subobject.)
- Your `Animal` subobject at offset *x* — is it a public base of yourself? (If so, delegate to `castToBase(mdo, typeid(Sponge))`.)

```
void *Reef_maybeFromHasAPublicChildOfTypeTo(void *mdo, int offset,
    const type_info& from, const type_info& to)
{
    if (offset == 0 && to == typeid(Fish)) return p + 0;
    // if (offset == 8 && to == typeid(Fish)) return nullptr;
    // if (offset == 8 && to == typeid(Sponge)) return nullptr;
    return nullptr;
}
```

Possible implementation

Again, the Itanium C++ ABI and the MSVC ABI take approaches that aren't this one. They use a graph of the class hierarchy, including "public/non-public" coloring of each edge, and they depth-first-search this graph on every `dynamic_cast`.

(And they do it wrong, with lots of bugs. Graph algorithms are hard.)

But let's put everything together and see what *we've* ended up with...

Possible implementation

```
template<class P, class From, class To = remove_pointer_t<P>>
To *dynamicast(From *p) {
    if constexpr (is_same_v<From, To>) {
        return p;
    } else if constexpr (is_base_of_v<To, From>) {
        return (To *)(p);
    } else if constexpr (is_void_v<To>) {
        return truly_dynamic_to_mdo(p);
    } else if constexpr (is_base_of_v<From, To>) {
        return truly_dynamic_from_base_to_derived<To>(p);
    } else {
        return truly_dynamic_between_unrelated_classes<To>(p);
    }
}
```

Technically we should also verify that `To` is an *accessible* base of `From` in the current lexical scope...

```
void *truly_dynamic_to_mdo(void *p)
{
    uint64_t *vptr =
        *reinterpret_cast<uint64_t **>(p);
    uint64_t mdoffset = vptr[-2];
    void *adjusted_this =
        static_cast<char *>(p) + mdoffset;
    return adjusted_this;
}

type_info& dynamic_typeid(void *p)
{
    return *(*reinterpret_cast<type_info ***>(p))[-1];
}
```

Fetch the vtable for “(some static type) in (most derived type).”

Get the “offset to MDO” from the vtable.

Get the MDO’s typeinfo from offset -1 in the vtable.

```
template<class To, class From>
To *truly_dynamic_from_base_to_derived(From *p) {
    void *mdo = truly_dynamic_to_mdo(p);
    const type_info& ti = dynamic_typeid(mdo);
    int offset = (char *)p - (char *)mdo;
    if (ti == From) {
        return nullptr;
    } else if (ti == To && ti.isPublicBaseOfMe(offset, From)) {
        return (To *)(mdo);
    } else if (void *so = ti.maybeFromHasAPublicChildOfTypeTo(
                              mdo, offset, From, To)) {
        return (To *)(so);
    } else if (ti.isPublicBaseOfMe(offset, From)) {
        return (To *)(ti.castToBase(mdo, To));
    }
    return nullptr;
}
```

Here, ***To*** represents the compile-time constant `typeid(To)`.

```
template<class To, class From>
To *truly_dynamic_between_unrelated_classes(From *p) {
    if constexpr (is_final_v<To> || is_final_v<From>) {
        return nullptr;
    }
    void *mdo = truly_dynamic_to_mdo(p);
    const type_info& ti = dynamic_typeid(mdo);
    int offset = (char *)p - (char *)mdo;
    if (ti == From) {
        return nullptr;
    } else if (ti == To) {
        return (To *)(mdo);
    } else if (ti.isPublicBaseOfMe(offset, From)) {
        return (To *)(ti.castToBase(mdo, To));
    }
    return nullptr;
}
```

Here, ***To*** represents the compile-time constant `typeid(To)`.

```
template<class P, class From, class To = remove_pointer_t<P>>
To *dynamicast(From *p) {
    if constexpr (is_same_v<From, To>) {
        return p;
    } else if constexpr (is_base_of_v<To, From>) {
        return (To *)(p);
    } else if constexpr (is_void_v<To>) {
        return truly_dynamic_to_mdo(p);
    } else if constexpr (is_base_of_v<From, To>) {
        return truly_dynamic_from_base_to_derived<To>(p);
    } else {
        return truly_dynamic_between_unrelated_classes<To>(p);
    }
}
```

Benchmark numbers

I wrote a Python script to generate random class hierarchies of 10 classes (of depth up to 7). I used [Google Benchmark](#) to measure `dynamic_cast` versus `dynamicast` for all possible combinations of `From*` and `To*`
(if the cast would be well-formed, not necessarily if it would return non-null).

The code is on my GitHub: http://github.com/Quuxplusone/from-scratch/

```
template<class To, class From, bool_if_t<can_dynamic_cast_v<From, To>> = true>
void run_benchmark(From *f) {
    To *p = dynamic_cast<To*>(f);
    benchmark::DoNotOptimize(p);
}

template<class To, class From, bool_if_t<!can_dynamic_cast_v<From, To>> = true>
void run_benchmark(From *) {}
```

Benchmark numbers

```
Benchmark                          Time           CPU Iterations
------------------------------------------------------------------
BM_native_void                   524 ns        522 ns    1312385
BM_native_Class1                1686 ns       1685 ns     393781
BM_native_Class2                2388 ns       2306 ns     287183
BM_native_Class3                2097 ns       2078 ns     339008
BM_native_Class4                3049 ns       3010 ns     246270
BM_native_Class5                3258 ns       3148 ns     235702
BM_dynamicast_void               973 ns        968 ns     731743
BM_dynamicast_Class1            1669 ns       1641 ns     400197
BM_dynamicast_Class2            2174 ns       2149 ns     307292
BM_dynamicast_Class3            1897 ns       1875 ns     363459
BM_dynamicast_Class4            2749 ns       2701 ns     260023
BM_dynamicast_Class5            3006 ns       2926 ns     236339
```

http://github.com/Quuxplusone/from-scratch/

Benchmark numbers (-O3)

```
Benchmark                           Time           CPU Iterations
------------------------------------------------------------------
BM_native_void                     64 ns         64 ns   10421164
BM_native_Class1                 2303 ns       2296 ns     319906
BM_native_Class2                 1941 ns       1937 ns     376849
BM_native_Class3                 1125 ns       1124 ns     641425
BM_native_Class4                 2503 ns       2488 ns     274518
BM_native_Class5                 2591 ns       2589 ns     272308
BM_dynamicast_void                124 ns        124 ns    5743071
BM_dynamicast_Class1              422 ns        420 ns    1608478
BM_dynamicast_Class2              416 ns        416 ns    1760156
BM_dynamicast_Class3              233 ns        232 ns    3067391
BM_dynamicast_Class4              564 ns        563 ns    1375597
BM_dynamicast_Class5              607 ns        606 ns    1084313
```

http://github.com/Quuxplusone/from-scratch/

Benchmark numbers (-O3 only for typeinfo)

```
Benchmark                         Time           CPU Iterations
----------------------------------------------------------------
BM_native_void                  459 ns        455 ns    1676245
BM_native_Class1               2491 ns       2412 ns     311500
BM_native_Class2               2470 ns       2377 ns     301865
BM_native_Class3               1453 ns       1452 ns     491394
BM_native_Class4               3245 ns       3109 ns     225830
BM_native_Class5               2933 ns       2924 ns     228102
BM_dynamicast_void              665 ns        663 ns    1107805
BM_dynamicast_Class1           1222 ns       1177 ns     640439
BM_dynamicast_Class2           1136 ns       1134 ns     664225
BM_dynamicast_Class3            793 ns        793 ns     922704
BM_dynamicast_Class4           1477 ns       1475 ns     505704
BM_dynamicast_Class5           1622 ns       1616 ns     424304
```

http://github.com/Quuxplusone/from-scratch/