

Why can't a derived class pointer point to a base class object without casting?

I have seen few Pet and Dog type examples for this type of basic question [here](#) and [here](#), but they do not make sense to me, here is why.

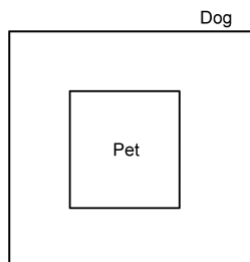
Suppose we have the following class structure

```
class Pet {};  
class Dog : public Pet {};
```

then the following statement

```
a (Dog) is a (Pet)
```

might be true in real life, **but is NOT true in C++**, in my opinion. Just look at the logical representation of a Dog object, it looks like this:



It is more appropriate to say

```
a (Dog) has a (Pet)
```

or

```
a (Pet) is a subset of (Dog)
```

which if you notice is a logical opposite of "a Dog is a Pet"

Now the problem is that #1 below is allowed while #2 is not:

```
Pet* p = new Dog; // [1] - allowed!  
Dog* d = new Pet; // [2] - not allowed without explicit casting!
```

My understanding is that [1] should not be allowed without warnings because there is no way a pointer should be able to point to an object of its superset's type (Dog object is a superset of Pet) simply because Pet does not know anything about the new members that Dog might have declared (the Dog - Pet subset in the diagram above).

[1] is equivalent of an `int*` trying to point to a `double` object!

Very obviously, I am missing a key point here which would turn my whole reasoning upside down. Can you please tell me what it is?

I believe making parallels to real world examples only complicate things. I would prefer to understand this in terms of technical details. Thanks!

[C++](#) [inheritance](#) [pointers](#)

edited May 23 '17 at 10:30

Community
1 • 1

asked Mar 17 '12 at 17:42

Lazer
31k • 85 • 229 • 320

4 I don't understand how you came to the statement "a (Pet) is a subset of (Dog)". Is your point that there are some dogs that aren't pets? – [Oliver Charlesworth](#) Mar 17 '12 at 17:45

No, my point is that the Dog object would always 'contain' Pet object members, and hence a `Pet*` would not be able to point to a `Dog` object (because there are a few members of Dog which are not Pet members). – [Lazer](#) Mar 17 '12 at 17:47

2 There are some pets that are not dogs - some people have cats or fish, too. – [Carl Norum](#) Mar 17 '12 at 17:48

2 @Lazer: Ah. Then you are conflating the **internal makeup** of an object (an object is made up of subobjects), and the **classification** (i.e. **class**) of an object (the set of all Dog objects is a subset of the set of all Pet objects). Inheritance relationships are concerned with the latter. When you say "has a" in your example above, what you really mean is "has the characteristics of". – [Oliver Charlesworth](#) Mar 17 '12 at 17:49

1 Your question seems to boil down to you taking "is a" as commutative. Well, it isn't. "A human is a bunch of atoms" is true, but "a bunch of atoms is a human" is obviously wrong in general. – [delnan](#) Mar 17 '12 at 17:56

11 Answers

Edit: Re-reading your question and my answer leads me to say this at the top:

Your understanding of `is a` in C++ (polymorphism, in general) is wrong.

A is B means A has at least the properties of B, possibly more, **by definition**.

This is compatible with your statements that a `Dog` has a `Pet` and that [the attributes of] a Pet is[are] a subset of [attributes] of `Dog`.

It's a matter of definition of polymorphism and inheritance. The diagrams you draw are aligned with the in-memory representation of instances of `Pet` and `Dog`, but are misleading in the way you interpret

The pointer `p` is defined to point to *any* `Pet`-compatible object, which in C++, is any subtype of `Pet` (Note: `Pet` is a subtype of itself by definition). The runtime is assured that, when the object behind `p` is accessed, it will contain whatever a `Pet` is expected to contain, **and possibly more**. The "possibly more" part is the `Dog` in your diagram. The way you draw your diagram lends to a misleading interpretation.

Think of the layout of class-specific members in memory:

```
Pet: [pet_data]
Dog: [pet_data][dog_data]
Cat: [pet_data][cat_data]
```

Now, whenever `Pet *p` points to, is required to have the `[pet_data]` part, and optionally, anything else. From the above listing, `Pet *p` may point to any of the three. As long you use `Pet *p` to access the objects, you may only access the `[pet_data]`, because you don't know what, if anything, is afterwards. It's a contract that says **This is at least a Pet, maybe more**.

Whatever `Dog *d` points to, must have the `[pet_data]` and `[dog_data]`. So the only object in memory it may point to, above, is the `Dog`. Conversely, through `Dog *d`, you may access both `[pet_data]` and `[dog_data]`. Similar for the `Cat`.

Let's interpret the declarations you are confused about:

```
Pet* p = new Dog; // [1] - allowed!
Dog* d = new Pet; // [2] - not allowed without explicit casting!
```

My understanding is that 1 should not be allowed without warnings because there is no way a pointer should be able to point to an object of its superset's type (Dog object is a superset of Pet) simply because Pet does not know anything about the new members that Dog might have declared (the Dog - Pet subset in the diagram above).

The pointer `p` expects to find `[pet_data]` at the location it points to. Since the right-hand-side is a `Dog`, and every `Dog` object has `[pet_data]` in front of its `[dog_data]`, pointing to an object of type `Dog` is perfectly okay.

The compiler doesn't know what **else** is behind the pointer, and this is why you cannot access `[dog_data]` through `p`.

The declaration is *allowed* because the presence of `[pet_data]` can be guaranteed by the compiler at compile-time. (this statement is obviously simplified from reality, to fit your problem description)

1 is equivalent of an `int*` trying to point to a double object!

There is no such subtype relationship between `int` and `double`, as is between `Dog` and `Pet` in C++. Try not to mix these into the discussion, because they are different: you cast between **values** of `int` and `double` (`(int) double` is explicit, `(double) int` is implicit), you cannot cast between **pointers to them**. Just forget this comparison.

As to [2]: the declaration states "`d` points to an object that has `[pet_data]` and `[dog_data]`, possibly more." But you are allocating only `[pet_data]`, so the compiler tells you you cannot do this.

In fact, the compiler cannot guarantee whether this is okay and it refuses to compile. There are legitimate situations where the compiler refuses to compile, but you, the programmer, know better. That's what `static_cast` and `dynamic_cast` are for. The simplest example in our context is:

```
d = p; // won't compile
d = static_cast<Dog*>(p); // [3]
d = dynamic_cast<Dog*>(p); // [4]
```

[3] will succeed always and lead to possibly hard-to-track bugs if `p` is not really a `Dog`.

[4] will will return `NULL` if `p` is not really a `Dog`.

I warmly suggest trying these casts out to see what you get. You should get garbage for `[dog_data]` from the `static_cast` and a `NULL` pointer for the `dynamic_cast`, assuming `RTTI` is enabled.

edited Mar 31 '16 at 12:19

answered Mar 17 '12 at 18:18

 Irly
5,905 ● 1 ● 24 ● 47

thanks for details, yours and amit's answers are exactly what I was looking for. But, why is the `int-double` example invalid? ideone.com/QRa3s - [Lazer](#) Mar 17 '12 at 18:27

Nowhere does the language specify that `int` is derived from `double` or vice-versa. This is the only reason. If they were defined as subclasses of one another, then pointers to them would be castable, but they are not. Try compiling `int *i; double *d; i = d; d = i;`, just won't work (without casting, which is, semantically, a different cast than what I wrote above, it's the equivalent of `reinterpret_cast` in this case). - [Irly](#) Mar 17 '12 at 18:35

Comparing `int` and `double` would be more like comparing `Dog` to a `Cat` which do not share any supertypes (like `Pet`), but have some common data. - [Irly](#) Mar 17 '12 at 18:37

@Irly a sentce in your answer: *The pointer p expects to find [pet_data] at the location it points to. Since the right-hand-side is a Dog, and every Dog object has [pet_data] in front of its [dog_data], pointing to an object of type Dog is perfectly okay.* Then assume a case there is another class `Animal` now `Pet:Animal` (inherits) and `Dog:Pet` (inherits) in this case it would be a `[Animal][Pet][Dog]`. In such case how can pointer to `Pet` point to `Dog`. - [sql_dumny](#) Aug 10 '16 at 12:17

The simplified memory layout of a `Pet` object will be `[animal_data][pet_data]`, and that of a `Dog` will be `[animal_data][pet_data][dog_data]` (you may be thinking that `Dog` would just have `[pet_data][dog_data]`, but that is incorrect - it inherits `Pet`'s supertypes as well). Now having cleared this up: a pointer to `Pet` will expect to see `[animal_data][pet_data]` wherever it points to. Since a `Dog` object fulfills that requirement, a `Pet` pointer may point to a `Dog` object. - [Irly](#) Aug 11 '16 at 12:25

A Dog is a Pet because it derives from class Pet. In C++ that pretty much fulfills the is-a requirement of OOP. [What is the Liskov substitution principle](#)


```
Dog* d = new Pet; // [2] - not allowed without explicit casting!
```

Of course it is not allowed, a Pet could just as well be a cat or a parrot.

edited May 23 '17 at 12:02

answered Mar 17 '12 at 17:47

 Community ♦
1 ● 1

 Bo Persson
72.4k ● 14 ● 104 ● 176

Thats what my point is. A Pet would never be a Cat or a Parrot. Rather, a parrot or a cat might contain Pet data members, which is perfectly fine. – [Lazer](#) Mar 17 '12 at 17:49

1 A cat can have a pet, but I bet that's unusual. – [Bo Persson](#) Mar 17 '12 at 17:52

Yes, in real life, but we are talking about C++! – [Lazer](#) Mar 17 '12 at 17:53

2 @Lazer Programming is modeling a reality for the computer to process, the fact that the `Dog` type contains a `Pet` sub object does not mean that `Pet` is a subset of `Dog`, a `Cat` is a `Pet` but not a `Dog`. You are mixing containment of a sub object with taxonomies of objects. You are focusing too much on the internal details of how it is implemented, and not enough on the interfaces. A `Pet` can be fed but cannot `bark`, a `Dog` can be `fed` (it behaves like a `Pet`) but can also `bark`. For all intents and purposes, a `Dog` behaves like a `Pet`. Read on Liskov! – [David Rodríguez - driebas](#) Mar 17 '12 at 17:57

It's an issue of hierarchical classification. If I tell my kids they can have a pet, then a dog is most certainly allowed. But if I tell them they can only have cat, then they cannot ask for a fish.

answered Mar 17 '12 at 17:48



[chrisaycock](#)

21.4k ● 7 ● 52 ● 88

I believe that is not how it works in C++, basically that is what my confusion is about. The Pet example does not apply to inheritance in C++ according to me. – [Lazer](#) Mar 17 '12 at 17:50

5 @Lazer No, this is *exactly* how it works in C++. `void foo(Pet*)` can take a `Dog*`, but `void bar(Cat*)` cannot take a `Fish*`. – [chrisaycock](#) Mar 17 '12 at 17:53

In terms of technical details:

The additional information of a `Dog` object is appended to the end of the `Pet` object, so the prefix [in bits] of `Dog` is actually a `Pet`, so there is no problems to assign a `Dog*` object to a `Pet*` variable. It is perfectly safe to access any of `Pet`'s fields/methods of the `Dog` object.

However - the opposite is not true. If you assign the address of the `Pet*` to a `Dog*` variable and then access one of the fields of `Dog` [which is not in `Pet`] you will get out of the allocated space.

Typing reasoning:

Also note that a value can be assigned to a variable only if it is of the correct type without casting [c++ is *static typing* language]. Since `Dog` is a `Pet`, `Dog*` is a `Pet*` - so this is not conflicting, but the other way around is not true.

edited Mar 17 '12 at 17:53

answered Mar 17 '12 at 17:47



[amit](#)

130k ● 16 ● 156 ● 263

1 This is the only answer so far that tells what I was looking for. But @amit, if we assign a `Dog*` object to a `Pet*` pointer, we will never be able to access some of the Dog members. Isn't that an information loss equivalent of converting a `double` to an `int`? – [Lazer](#) Mar 17 '12 at 17:57

@Lazer: You will never be able to access the `Dog` members through **that particular pointer**. Nothing has been lost, you're simply expressing the fact that you're happy to work with a generic `Pet`. – [Oliver Charlesworth](#) Mar 17 '12 at 17:58

There's no information lost, you just can't access those members. If you are referring to a particular `Dog` object as a `Pet`, all you know is that it's a `Pet`. You don't know if it's a `Dog` or a `Cat` or a `NorwegianBlueParrot`. If you want to work with the `Dog` interface, just use a `Dog` variable - problem solved. – [Carl Norum](#) Mar 17 '12 at 17:59

@OliCharlesworth: Fair enough. But, this is why I think example 2 in my question (`Dog* d = new Pet;`) is perfectly okay, because we can access all the members of our Pet object! – [Lazer](#) Mar 17 '12 at 18:00

@Lazer, but what if you try to access a `Dog`-specific member? CRASH. You can't tell someone "Hey, here's a `Dog` object" and not allow them to call `Dog` methods. – [Carl Norum](#) Mar 17 '12 at 18:01

I think you are confusing what *is-a* is intended to mean in the OO context. You can say that a `Dog` has a `Pet` sub object, and that is true if you go down to the bit representation of the objects. But the important thing is that programming is about modeling a reality into a program that a computer can process. Inheritance is the way that you model the relationship *is-a*, as per your example:

A `Dog` *is-a* `Pet`.

In common parlance means that it exhibits all behaviors of a Pet, with maybe some characteristic behaviors that differ (barks) but it is an animal and it provides company, you can feed it... All those behaviors would be defined by (virtual) member functions in the `Pet` class and might be overridden in the `Dog` type, as other operations are defined. But the important part is that by using inheritance all instances of `Dog` can be used where an instance of `Pet` is required.

answered Mar 17 '12 at 17:52



[David Rodríguez - driebas](#)

165k ● 14 ● 217 ● 417

You've gotten confused between base and parent classes.

`Pet` is the base class. A `Pet*` could point any number of different types, so long as they inherit from `Pet`. So it's no surprise that `Pet* pet = new Dog` is allowed. `Dog` is a `Pet`. I have a pointer to a `Pet`, which happens to be a `Dog`.

On the other hand, if I have a `Pet*`, I have no idea what it actually points to. It could point to a `Dog`, but it could also point to a `Cat`, a `Fish`, or something else entirely. As such, the language won't let me call `Pet->bark()` because not all `Pet`s can `bark()` (`Cat` `s` `meow()`, for instance).

If, however, I have a `Pet*` that I **know** is, in fact, a `Dog`, then it's entirely safe to cast to a `Dog`, and then call `bark()`.

So:

```
Pet* p = new Dog; // sure this is allowed, we know that all Dogs are Pets
Dog* d = new Pet; // this is not allowed, because Pet doesn't support everything
Dog does
```

answered Mar 17 '12 at 18:13

 [Max Lybbert](#)
16k • 3 • 33 • 64

In English, the statements you're trying to work out might be 'The aspects of a dog which cause it to be a pet are a subset of all the aspects of a dog' and 'the set of all entities which are dogs are a subset of the set of entities which are pets'.

D, P such that $D(x) \Rightarrow x \in \text{Dog}$, $P(x) \Rightarrow x \in \text{Pet}$

$D(x) \Rightarrow P(x)$

($D(x)$ is true if x has all the aspects of a dog, so this is saying that the aspects of a thing which is a dog are a super set of the aspects of things which are pets - $P(x)$ is true if $D(x)$ is true, but not necessarily the other way round)

$\text{Dog} \supseteq \text{Pet} \Rightarrow$

$\forall x \ x \in \text{Dog} \Rightarrow x \in \text{Pets}$ (every dog is a pet)

But if $D(x) \equiv x \in \text{Dog}$, then these are the same statement.

So saying 'the aspects of a Dog which make it a pet are a subset of the dog as a whole' is equivalent to saying 'the set of dogs is a subset of the set of pets'

answered Mar 17 '12 at 18:11

 [Pete Kirkham](#)
41.5k • 3 • 75 • 146

Consider this scenario (sorry for such a cheesy example):

A vehicle can be any vehicle

```
class Vehicle
{
    int numberOfWheels;
    void printWheels();
};
```

A Car is a vehicle

```
class Car: public Vehicle
{
    // the wheels are inherited
    int numberOfDoors;
    bool doorsOpen;
    bool isHatchBack;
};
```


A bike is a vehicle, but this vehicle is not a Car which is a vehicle too

```
class Bike: public Vehicle
{
    int numberOfWings; // sorry, don't know exact name
    // the wheels are inherited
};
```

So I hope not only you can see real life difference, but also notice that the memory layout in the program will be different for `Bike` and `Car` objects even though **they're** both `Vehicles`. That's why a child object cannot be any type of child; it can be only what was defined to be.

edited Mar 17 '12 at 18:25

answered Mar 17 '12 at 18:17

 [unexplored](#)
747 • 3 • 10 • 23

All the answers above are good. I just wanna add one more thing. I think your diagram of dog/pet is misleading.

I understand why you sketched the DOG diagram as a superset of the PET one: you probably thought that, since a Dog has more attributes than a pet, it needs to be represented by a bigger set.

However, whenever you draw a diagram where a set B is a subset of a set A, you are saying that the number of objects of type B is for sure NO MORE than the objects of type A. On the other hand, since the objects of type B have more properties, you are allowed to do more operations on them, since you can do ALL the operations allowed on objects of type A PLUS some more.

If you happen to know something about functional analysis (it's a long shot, but maybe you saw it), it's the same relation that exists between banach spaces and their duals: the smaller the space, the bigger the set of operation you can do on them.

answered Mar 17 '12 at 21:22

 [bartgol](#)
635 • 9 • 19

The actual reason is - a derived class has all information about a base class and also some extra bit of information. Now a pointer to a derived class will require more space and that is not sufficient in base class. So the a pointer to a derived class cannot point to it. While on the other hand the reverse is true.

answered Jun 22 '13 at 7:58

 [tanu](#)
1

Based on some of previous answers I developed some understanding and I am putting them in my words.

5)Base class pointer variable can point to derived class object. But not vice versa.

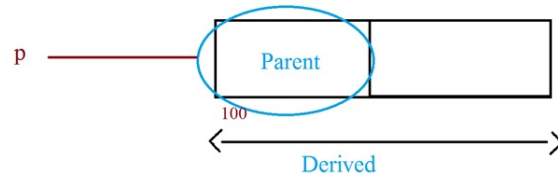
It is a very important to note with base class pointer variable pointed to derived class you can access only base class properties, not derived class properties because:

```
Parent *p;
```

```
Derived d;
```

```
//&d returns starting address of d;
```

```
p=&d; //now p just eats the parent properties that are sitting at starting address of d
```



answered Aug 10 '16 at 14:45

 [sql_dummy](#)
207 ● 2 ● 13