Why can't I pass a reference to a derived class to a function that takes a reference to a base class by reference?

devblogs.microsoft.com/oldnewthing/20090812-00

August 12, 2009

Raymond Chen

“Why can’t I pass a reference to a derived class to a function that takes a reference to a base class by reference?” That’s a confusing question, but it’s phrased that way because the simpler phrasing is wrong!

Ths misleading simplified phrasing of the question is “Why can’t I pass a reference to a derived class to a function that takes a base class by reference?” And in fact the answer is “You can!”

```
class Base { }
class Derived : Base { }
class Program {
  static void f(Base b) { }
  public static void Main()
  {
      Derived d = new Derived();
      f(d);
  }
}
```

Our call to `f` passes a reference to the derived class to a function that takes a reference to the base class. This is perfectly fine.

When people ask this question, they are typically wondering about passing a reference to the base class *by reference*. There is a double indirection here. You are passing a reference to a variable, and the variable is a reference to the base class. And it is this double reference that causes the problem.

```
class Base { }
class Derived : Base { }
class Program {
  static void f(ref Base b) { }
  public static void Main()
  {
      Derived d = new Derived();
      f(ref d); // error
  }
}
```

Adding the `ref` keyword to the parameter results in a compiler error:

```
error CS1503: Argument '1': cannot convert from 'ref Derived' to 'ref Base'
```

The reason this is disallowed is that it would allow you to violate the type system. Consider:

```
  static void f(ref Base b) { b = new Base(); }
```

Now things get interesting. Your call to `f(ref d)` passes a reference to a `Derived` by reference. When the `f` function modifies its formal parameter `b`, it's actually modifying your variable `d`. What's worse, it's putting a `Base` in it! When `f` returns, your variable `d`, which is declared as being a reference to a `Derived` is actually a reference to the base class `Base`.

At this point everything falls apart. Your program calls some method like `d.OnlyInDerived()`, and the CLR ends up executing a method on an object that doesn't even support that method.

You actually knew this; you just didn't know it. Let's start from the easier cases and work up. First, passing a reference into a function:

```
void f(SomeClass s);
...
   T t = new T();
   f(t);
```

The function `f` expects to receive a reference to a `SomeClass`, but you're passing a reference to a `T`. When is this legal?

"Duh. `T` must be `SomeClass` or a class derived from `SomeClass`."

What's good for the goose is good for the gander. When you pass a parameter as `ref`, it not only goes into the method, but it also comes out. (Not strictly true but close enough.) You can think of it as a bidirectional parameter to the function call. Therefore, the rule "If a function expects a reference to a class, you must provide a reference to that class or a derived class" applies in both directions. When the parameter goes in, you must provide a reference to that

class or a derived class. And when the parameter comes out, it also must be a reference to that class or a derived class (because the function is “passing the parameter” back to you, the caller).

But the only time that `S` can be `T` or a subclass, while simultaneously having `T` be `S` or a subclass is when `S` and `T` are the same thing. This is just the law of antisymmetry for partially-ordered sets: “if $a \le b$ and $b \le a$, then a = b.”

Raymond Chen

Follow