Why does C++ think my class is copy-constructible when it can't be copy-constructed?

devblogs.microsoft.com/oldnewthing/20250606-00/?p=111254

June 6, 2025

Consider the following scenario:

```
template<typename T>
struct Base
{
    // Default-constructible
    Base() = default;

    // Not copy-constructible
    Base(Base const &) = delete;
};

template<typename T>
struct Derived : Base<T>
{
    Derived() = default;
    Derived(Derived const& d) : Base<T>(d) {}
};

// This assertion passes?
static_assert(
    std::is_copy_constructible_v<Derived<int>>);
```

Why does this assertion pass? It is plainly evident that you cannot copy a `Derived<int>` because doing so will try to copy the `Base<int>`, which is not copyable. Indeed, if you try to copy it, you get an error:

```
void example(Derived<int>& d)
{
    Derived<int> d2(d);
    // msvc: error C2280: 'Base<T>::Base(const Base<T> &)':
    //       attempting to reference a deleted function
    // gcc:  error: use of deleted function 'Base<T>::Base(const Base<T>&)
    //       [with T = int]'
    // clang: error: call to deleted constructor of 'Base<int>'
}
```

Okay, so the compiler thinks that `Derived<int>` is copy-constructible, but then when we try to do it, we find out that it isn't!

What's going on is that the compiler is determining copy-constructibility by checking whether the class has a non-deleted copy constructor. And in the case of `Derived<T>` it does haev a non-deleted copy constructor. You declared it yourself!

```
    Derived(Derived const& d) : Base<T>(d) {}
```

So yes, there is a copy constructor. It can't be instantiated, but the compiler doesn't care. It is going based on what you tell it, and you told it that you can copy it.

After all, another possibly copy constructor would have been

```
    Derived(Derived const& d) : Base<T>() {}
```

and this one instantiates successfully. Copying a `Derived` default-constructs the `Base` base class rather than copy-constructing it.

Imagine that we moved the definition out of line.

```
template<typename T>
struct Derived : Base<T>
{
    Derived() = default;
    Derived(Derived const& d);
};
```

What should the answer to the question "Is this copy-constructible?" be? You don't know what the definition is, only its declaration. Should the compiler halt compilation with the error message "Unable to predict the future"? But what if you didn't want the expose the implementation of the copy constructor in the header file?

The rule for determining copy constructibility is whether a non-deleted copy constructor is present. In the case of `Derived`, it is present. It may not be instantiatable, but that's not what `is_copy_constructible` looks for.[1]

Now, non-copyability inherits by default, so we could have just allowed the copy constructor to be defaulted:

```
template<typename T>
struct Derived : Base<T>
{
    Derived() = default;
    Derived(Derived const& d) = default;
};
```

The implicitly-defined or explicitly-defaulted copy constructor is defined as deleted if any base class is not copy-constructible, in which case the declaration is treated as if it had said `= delete`. That `= delete` can be detected by `is_copy_constructible` and result in the assertion failing.

But if you come out and make a custom copy constructor that is not deleted, the compiler assumes you will make good on your promise.

Related reading: Why does `std::is_copy_constructible` report that a vector of move-only objects is copy constructible?

[1] Requiring that the type be complete *and all members defined* is not a reasonable requirement because that would require definitions of all class methods to be present in header files. Your entire program has been reduced to a header-only project.