The C++ implicit assignment operator is a non-ref-qualified member, even if the base class’s assignment has a ref-qualifier

devblogs.microsoft.com/oldnewthing/20210916-00

September 16, 2021

Raymond Chen

Consider the following C++ class:

```
struct Base
{
    Base& operator=(Base const&) & = default;

    void BaseMethod();
};
```

This defines a class for which you can assign to an lvalue reference, but not to an rvalue reference.

```
extern Base GetBase();

Base b;
b = GetBase(); // allowed
GetBase() = b; // not allowed
```

Assigning to an rvalue is not generally useful, since the object has no name, and consequently it is difficult to do anything with the assigned-to object afterward.[1]

Great, we got rid of assignment to a temporary, which we’ve seen has been a source of confusion.

Now consider this:

```
struct Derived : Base
{
};

Derived d;
Derived() = d; // is this allowed?
```

We created a derived class and inherited the assignment operator from it. Do you expect the inherited assignment operator to block rvalues?

You probably guessed that the answer is *no*, seeing as I gave it away in the title.

The reason is that I lied when I said that the assignment operator was inherited. It was not inherited. It was *implicitly declared*.

The rules for implicit declaration of the copy assignment operator are spelled out in **[class.copy.assign]**, paragraphs 2 and 4. The short version is that a class is eligible for an implicitly-declared copy assignment operator if its base classes and non-static members all have a copy assignment operator. (Analogous rules apply for the implicitly-declared move assignment operator.)

The catch is that the implicitly-declared copy assignment and move assignment operators are declared as an *unqualified* assignment operator, regardless of the reference-qualifications of the base classes and members. In our example, we get

```
struct Derived : Base
{
    // compiler autogenerates these
    Derived& operator=(Derived const&) = default;
    //                                ^ no &
};
```

The lack of a ref-qualification means that this assignment operator applies equally to lvalues and rvalues.

Our attempt to block rvalue assignment fails to propagate to derived classes!

In order to repair this, each derived class must redeclare its assignment operator as lvalue-only.

```
struct Derived : Base
{
    Derived& operator=(Derived const&) & = default;
};
```

Oh, we've only started our journey down the rabbit-hole.

At least for now, explicitly declaring a copy assignment operator does not cause the implicitly-declared copy/move constructors to disappear, but the behavior is noted as deprecated in the C++ language specification, with the note that a future version of the language may indeed delete them.

```
Derived d;
Derived d2{ d }; // on borrowed time
```

To make sure you don’t run into trouble in the future, you’ll want to declare them explicitly.

```
struct Derived : Base
{
    Derived(Derived const&) = default;
    Derived(Derived&&) = default;
    Derived& operator=(Derived const&) & = default;
};
```

Great, we’ve restored the copy and move constructors.

But explicitly declaring any constructors causes us to lose the implicitly-declared default constructor.

```
Derived d; // doesn't work any more
```

We’ll have to bring that back too.

```
struct Derived : Base
{
    Derived() = default;
    Derived(Derived const&) = default;
    Derived(Derived&&) = default;
    Derived& operator=(Derived const&) & = default;
};
```

The same exercise applies if we also want to block the move assignment operator to rvalues, but it’s more urgent because an explicit declaration of a move assignment operator does delete both the copy and move constructors even in C++20.

```
struct Base
{
    Base& operator=(Base const&) & = default;
    Base& operator=(Base&&) & = default;

    void BaseMethod();
};

struct Derived : Base
{
    Derived() = default;
    Derived(Derived const&) = default;
    Derived(Derived&&) = default;
    Derived& operator=(Derived const&) & = default;
    Derived& operator=(Derived&&) & = default;
};
```

Phew, that was annoying.

[1] I mean, I guess you could do this:

```
Base b;

Something(GetBase() = b);
(GetBase() = b).BaseMethod();
```

but it seems pointless to go to the effort of asking `GetBase` to create you a `Base` object, only to overwrite it with your own. You may as well just create your own temporary.

```
Something(Base(b));
Base(b).BaseMethod();
```

Or, if you didn’t even mean to create a temporary, just use the original value:

```
Something(b);
b.BaseMethod();
```

Raymond Chen

Follow