Why does unique_ptr<Derived> implicitly cast to unique_ptr<Base>?

Asked 4 years, 7 months ago Modified 4 years, 6 months ago Viewed 2k times



I wrote the following code that uses unique_ptr<Derived> where a unique_ptr<Base> is expected

23



```
class Base {
    int i;
 public:
    Base( int i ) : i(i) {}
    int getI() const { return i; }
class Derived : public Base {
    float f;
 public:
    Derived( int i, float f ) : Base(i), f(f) {}
    float getF() const { return f; }
};
void printBase( unique_ptr<Base> base )
    cout << "f: " << base->qetI() << endl;</pre>
}
unique_ptr<Base> makeBase()
    return make_unique<Derived>( 2, 3.0f );
}
unique_ptr<Derived> makeDerived()
    return make_unique<Derived>( 2, 3.0f );
int main( int argc, char * argv [] )
    unique_ptr<Base> base1 = makeBase();
    unique_ptr<Base> base2 = makeDerived();
    printBase( make_unique<Derived>( 2, 3.0f ) );
    return 0;
}
```

and i expected this code to not compile, because according to my understanding unique_ptr<Base> and unique_ptr<Derived> are unrelated types and unique_ptr<Derived> isn't in fact derived from unique_ptr<Base> so the assignment shouldn't work.

But thanks to some magic it works, and i don't understand why, or even if it's safe to do so. Can someone explain please?

smart pointers are to enrich what pointers can do not to limit it. If this wasnt possible unique_ptr would be rather useless in the presence of inheritance – 463035818_is_not_an_ai Oct 1, 2019 at 8:49

4 — "But thanks to some magic it works". Nearly, you got UB as Base doesn't have virtual destructor. – Jarod42 Oct 1, 2019 at 12:43

3 Answers

Sorted by: Highest s

Highest score (default)

\$



The bit of magic you're looking for is the converting constructor #6 here:

28

```
template<class U, class E>
unique_ptr(unique_ptr<U, E> &&u) noexcept;
```





It enables constructing a std::unique_ptr<T> implicitly from an expiring std::unique_ptr<U> if (glossing over deleters for clarity):



unique_ptr<U, E>::pointer is implicitly convertible to pointer



Which is to say, it mimicks implicit raw pointer conversions, including derived-to-base conversions, and does what you expect™ safely (in terms of lifetime – you still need to ensure that the base type can be deleted polymorphically).

Share Edit Follow Flag

edited Oct 1, 2019 at 10:07

answered Oct 1, 2019 at 8:50



Quentin 62.6k 7 135 196



AFAIK the deleter of Base won't call the destructor of Derived, so I'm not sure whether it's really safe. (It's no less safe than raw pointer, admittedly.) – cpplearner Oct 1, 2019 at 10:02



Because std::unique.ptr has a converting constructor as



template< class U, class E >
unique_ptr(unique_ptr<U, E>&& u) noexcept;



and



This constructor only participates in overload resolution if all of the following is true:

a) unique_ptr<U, E>::pointer is implicitly convertible to pointer

...

A Derived* could convert to Base* implicitly, then the converting constructor could be applied for this case. Then a std::unique_ptr<Base> could be converted from a std::unique_ptr<Derived> implicitly just as the raw pointer does. (Note that the std::unique_ptr<Derived> has to be an rvalue for constructing std::unique_ptr<Base> because of the characteristic of std::unique_ptr.)

Share Edit Follow Flag

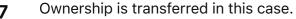
edited Oct 2, 2019 at 2:11

answered Oct 1, 2019 at 8:48





You can implicitly construct a std::unique_ptr<T> instance from an rvalue of std::unique_ptr<S> whenever S is convertible to T. This is due to constructor #6 here.





In your example, you have only rvalues of type std::uinque_ptr<Derived> (because the return value of std::make_unique is an rvalue), and when you use that as a std::unique_ptr<Base>, the constructor mentioned above is invoked. The

std::unique_ptr<Derived> objects in question hence only live for a short amount of time, i.e. they are created, then ownership is passed to the std::unique_ptr<Base> object that is used further on.

Share Edit Follow Flag

answered Oct 1, 2019 at 8:50



37.9k 3 69 119