# Sometimes perfect forwarding can be too perfect: Lazy conversion is lazy

devblogs.microsoft.com/oldnewthing/20221123-00

November 23, 2022

Raymond Chen

C++11 introduced the concept of *perfect forwarding*, which uses *universal references forwarding references* to capture the value category of a passed-in parameter, so that it can be passed to another function as the same kind of reference.

Forwarding references are used in many places, but the trap I'm looking at today is when the forwarding reference causes a conversion to be delayed to a point where it is no longer possible.

```
class Base
{
    /* ... */
};

class BaseAcceptor
{
public:
    BaseAcceptor(Base* base);
    /* ... */
};

class Derived : Base
{
    void DoSomething()
    {
        auto p1 = std::unique_ptr<BaseAcceptor>(new BaseAcceptor(this));
        auto p2 = std::shared_ptr<BaseAcceptor>(new BaseAcceptor(this));

        std::vector<BaseAcceptor> v;
        v.push_back(BaseAcceptor(this));
    }

    /* ... */
};
```

But then you realize that you should be using the `make_` functions, so that you can avoid having to write the `new` keyword, and you can use the `emplace` alternative to construct the vector element in place.

```
class Derived : Base
{
    void DoSomething()
    {
        auto p1 = std::make_unique<BaseAcceptor>(this);
        auto p2 = std::make_shared<BaseAcceptor>(this);

        std::vector<BaseAcceptor> v;
        v.emplace_back(this);
    }
};
```

And then everything explodes into tiny little pieces.

```
// clang

In file included from memory:76:
unique_ptr.h:1072:38: error: cannot cast 'Derived' to its private base class 'Base'
    { return unique_ptr<_Tp>(new _Tp(std::forward<_Args>(__args)...)); }
                                     ^
test.cpp:19:24: note: in instantiation of function template specialization
'std::make_unique<BaseAcceptor, Derived *>' requested here
        auto p1 = std::make_unique<BaseAcceptor>(this);
                       ^
test.cpp:14:17: note: implicitly declared private here
class Derived : Base
                ^~~~

In file included from vector:62:
stl_construct.h:119:29: error: cannot cast 'Derived' to its private base class 'Base'
      ::new((void*)__p) _Tp(std::forward<_Args>(__args)...);
                            ^
alloc_traits.h:635:9: note: in instantiation of function template specialization
'std::_Construct<BaseAcceptor, Derived *>' requested here
        { std::_Construct(__p, std::forward<_Args>(__args)...); }
               ^
shared_ptr_base.h:604:30: note: in instantiation of function template specialization
'std::allocator_traits<std::allocator<void>>::construct<BaseAcceptor, Derived *>'
requested here
          allocator_traits<_Alloc>::construct(__a, _M_ptr(),
                                    ^
shared_ptr_base.h:972:6: note: in instantiation of function template specialization
'std::_Sp_counted_ptr_inplace<BaseAcceptor, std::allocator<void>,
__gnu_cxx::_S_atomic>::_Sp_counted_ptr_inplace<Derived *>' requested here
            _Sp_cp_type(__a._M_a, std::forward<_Args>(__args)...);
            ^
shared_ptr_base.h:1712:14: note: in instantiation of function template specialization
'std::__shared_count<__gnu_cxx::_S_atomic>::__shared_count<BaseAcceptor,
std::allocator<void>, Derived *>' requested here
        : _M_ptr(), _M_refcount(_M_ptr, __tag, std::forward<_Args>(__args)...)
                    ^
shared_ptr.h:464:4: note: in instantiation of function template specialization
'std::__shared_ptr<BaseAcceptor,
__gnu_cxx::_S_atomic>::__shared_ptr<std::allocator<void>, Derived *>' requested here
        : __shared_ptr<_Tp>(__tag, std::forward<_Args>(__args)...)
          ^
shared_ptr.h:1009:14: note: in instantiation of function template specialization
'std::shared_ptr<BaseAcceptor>::shared_ptr<std::allocator<void>, Derived *>'
requested here
      return shared_ptr<_Tp>(_Sp_alloc_shared_tag<_Alloc>{__a},
             ^
test.cpp:20:24: note: in instantiation of function template specialization
'std::make_shared<BaseAcceptor, Derived *>' requested here
        auto p2 = std::make_shared<BaseAcceptor>(this);
                       ^
test.cpp:14:17: note: implicitly declared private here
```

```
class Derived : Base
                ^~~~

In file included from vector:61:
In file included from allocator.h:46:
In file included from c++allocator.h:33:
new_allocator.h:182:27: error: cannot cast 'Derived' to its private base class 'Base'
        { ::new((void *)__p) _Up(std::forward<_Args>(__args)...); }
                                 ^
alloc_traits.h:516:8: note: in instantiation of function template specialization
'std::__new_allocator<BaseAcceptor>::construct<BaseAcceptor, Derived *>' requested
here
          __a.construct(__p, std::forward<_Args>(__args)...);
              ^
vector.tcc:117:21: note: in instantiation of function template specialization
'std::allocator_traits<std::allocator<BaseAcceptor>>::construct<BaseAcceptor, Derived
*>' requested here
            _Alloc_traits::construct(this->_M_impl, this->_M_impl._M_finish,
                           ^
test.cpp:23:11: note: in instantiation of function template specialization
'std::vector<BaseAcceptor>::emplace_back<Derived *>' requested here
        v.emplace_back(this);
          ^
test.cpp:14:17: note: implicitly declared private here
class Derived : Base
                ^~~~

// gcc
In file included from memory:76,
                 from test.cpp:2:
test.cpp std::make_unique(_Args&& ...) [with _Tp = BaseAcceptor; _Args = {Derived*};
__detail::__unique_ptr_t<_Tp> = __detail::__unique_ptr_t<BaseAcceptor>]':
test.cpp:19:49:   required from here
unique_ptr.h:1072:30: error: 'Base' is an inaccessible base of 'Derived'
 1072 |     { return unique_ptr<_Tp>(new _Tp(std::forward<_Args>(__args)...)); }
      |                              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
In file included from c++allocator.h:33,
                 from allocator.h:46,
                 from vector:61,
                 from test.cpp:1:
new_allocator.h: In instantiation of 'void std::__new_allocator<_Tp>::construct(_Up*,
_Args&& ...) [with _Up = BaseAcceptor; _Args = {Derived*}; _Tp = BaseAcceptor]':
alloc_traits.h:516:17:   required from 'static void
std::allocator_traits<std::allocator<_Tp1> >::construct(allocator_type&, _Up*,
_Args&& ...) [with _Up = BaseAcceptor; _Args = {Derived*}; _Tp = BaseAcceptor;
allocator_type = std::allocator<BaseAcceptor>]'
vector.tcc:117:30:   required from 'std::vector<_Tp, _Alloc>::reference
std::vector<_Tp, _Alloc>::emplace_back(_Args&& ...) [with _Args = {Derived*}; _Tp =
BaseAcceptor; _Alloc = std::allocator<BaseAcceptor>; reference = BaseAcceptor&]'
required from here
new_allocator.h:182:11: error: 'Base' is an inaccessible base of 'Derived'
  182 |         { ::new((void *)__p) _Up(std::forward<_Args>(__args)...); }
```

```
      |           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
In file included from vector:62:
stl_construct.h: In instantiation of 'void std::_Construct(_Tp*, _Args&& ...) [with
_Tp = BaseAcceptor; _Args = {Derived*}]':
alloc_traits.h:635:19:   required from 'static void
std::allocator_traits<std::allocator<void> >::construct(allocator_type&, _Up*,
_Args&& ...) [with _Up = BaseAcceptor; _Args = {Derived*}; allocator_type =
std::allocator<void>]'
shared_ptr_base.h:604:39:   required from 'std::_Sp_counted_ptr_inplace<_Tp, _Alloc,
_Lp>::_Sp_counted_ptr_inplace(_Alloc, _Args&& ...) [with _Args = {Derived*}; _Tp =
BaseAcceptor; _Alloc = std::allocator<void>; __gnu_cxx::_Lock_policy _Lp =
__gnu_cxx::_S_atomic]'
shared_ptr_base.h:971:16:   required from
'std::__shared_count<_Lp>::__shared_count(_Tp*&, std::_Sp_alloc_shared_tag<_Alloc>,
_Args&& ...) [with _Tp = BaseAcceptor; _Alloc = std::allocator<void>; _Args =
{Derived*}; __gnu_cxx::_Lock_policy _Lp = __gnu_cxx::_S_atomic]'
shared_ptr_base.h:1712:14:   required from 'std::__shared_ptr<_Tp,
_Lp>::__shared_ptr(std::_Sp_alloc_shared_tag<_Tp>, _Args&& ...) [with _Alloc =
std::allocator<void>; _Args = {Derived*}; _Tp = BaseAcceptor; __gnu_cxx::_Lock_policy
_Lp = __gnu_cxx::_S_atomic]'
shared_ptr.h:464:59:   required from
'std::shared_ptr<_Tp>::shared_ptr(std::_Sp_alloc_shared_tag<_Tp>, _Args&& ...) [with
_Alloc = std::allocator<void>; _Args = {Derived*}; _Tp = BaseAcceptor]'
shared_ptr.h:1009:14:   required from 'std::shared_ptr<typename std::enable_if<(!
std::is_array< <template-parameter-1-1> >::value), _Tp>::type>
std::make_shared(_Args&& ...) [with _Tp = BaseAcceptor; _Args = {Derived*}; typename
enable_if<(! is_array< <template-parameter-1-1> >::value), _Tp>::type =
BaseAcceptor]'
test.cpp:20:49:   required from here
stl_construct.h:119:7: error: 'Base' is an inaccessible base of 'Derived'
  119 |       ::new((void*)__p) _Tp(std::forward<_Args>(__args)...);
      |       ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// msvc
memory(1630): error C2243: 'type cast': conversion from 'Derived *const ' to 'Base *'
exists, but is inaccessible
test.cpp(19): note: see reference to function template instantiation
'std::unique_ptr<BaseAcceptor,std::default_delete<_Ty>>
std::make_unique<BaseAcceptor,Derived*const >(Derived *const &&)' being compiled
        with
        [
            _Ty=BaseAcceptor
        ]

vector(1604): warning C4530: C++ exception handler used, but unwind semantics are not
enabled. Specify /EHsc
vector(1601): note: while compiling class template member function 'void
std::vector<BaseAcceptor,std::allocator<_Ty>>::_Reallocate(unsigned __int64)'
        with
        [
            _Ty=BaseAcceptor
        ]
```

```
vector(1631): note: see reference to function template instantiation 'void
std::vector<BaseAcceptor,std::allocator<_Ty>>::_Reallocate(unsigned __int64)' being
compiled
        with
        [
            _Ty=BaseAcceptor
        ]
test.cpp(22): note: see reference to class template instantiation
'std::vector<BaseAcceptor,std::allocator<_Ty>>' being compiled
        with
        [
            _Ty=BaseAcceptor
        ]

memory(901): error C2243: 'type cast': conversion from 'Derived *const ' to 'Base *'
exists, but is inaccessible
memory(971): note: see reference to function template instantiation
'std::_Ref_count_obj<_Ty>::_Ref_count_obj<Derived*const >(Derived *const &&)' being
compiled
        with
        [
            _Ty=BaseAcceptor
        ]
memory(971): note: see reference to function template instantiation
'std::_Ref_count_obj<_Ty>::_Ref_count_obj<Derived*const >(Derived *const &&)' being
compiled
        with
        [
            _Ty=BaseAcceptor
        ]
test.cpp(20): note: see reference to function template instantiation
'std::shared_ptr<BaseAcceptor> std::make_shared<BaseAcceptor,Derived*const >(Derived
*const &&)' being compiled
```

What does all this mean?

The error boils down to this:

```
// clang
error: cannot cast 'Derived' to its private base class 'Base'

// gcc
error: 'Base' is an inaccessible base of 'Derived'

// msvc
error C2243: 'type cast': conversion from 'Derived *const ' to 'Base *' exists, but
is inaccessible
```

When `Derived::DoSomething` constructs the `BaseAcceptor` class, it passes its own `this` as the constructor parameter. Inside that class's own method, `this` gives you a `Derived*`.[1] However, the constructor of `BaseAcceptor` takes a `Base*`. Fortunately, there is a conversion from `Derived*` to `Base*` available to `Derived`, and the compiler performs that conversion in order to construct the `BaseAcceptor`.

On the other hand, if you use `make_unique`, `make_shared`, `emplace_*`, or other functions that perfect-forward their parameters, the parameter is passed as a `Derived*`. That `Derived*` is then forwarded perfectly to the `BaseAcceptor` constructor, and the compiler realizes, "Oh, wait, `BaseAcceptor`'s constructor doesn't accept a `Derived*`. Let me see if I can convert it." And it can't because the `Derived*`-to-Base* conversion is not available to `make_unique`.

The conversion is not available because `Base` is a private base class of `Derived`. Only `Derived` itself is allowed to access the `Base` portion of `Derived`, and that includes producing a pointer to it.

The goal of perfect forwarding is to keep all the parameters in their original form, so that they reach their destination in their original form, without triggering a decay or copy or conversion or any other nonsense along the way.

That's great, but it also means that the conversion is deferred. What leaves the friendly confines of the `Derived` class is a `Derived*`, and when it is finally used to construct a `BaseAcceptor`, the desired conversion is no longer available.

Okay, now that we understand the problem, how do we fix it?

One way is to force the conversion while still inside the `Derived` class. That way, the conversion happens while it is still accessible.

```
class Derived : Base
{
    void DoSomething()
    {
        auto p1 = std::make_unique<BaseAcceptor>(static_cast<Base*>(this));
        auto p2 = std::make_shared<BaseAcceptor>(static_cast<Base*>(this));

        std::vector<BaseAcceptor> v;
        v.emplace_back(static_cast<Base*>(this));
    }
};
```

Another way is to make the conversion public.

```
class Derived : public Base
{
    void DoSomething()
    {
        auto p1 = std::make_unique<BaseAcceptor>(this);
        auto p2 = std::make_shared<BaseAcceptor>(this);

        std::vector<BaseAcceptor> v;
        v.emplace_back(this);
    }
};
```

In this specific case, the private-ness of the `Base` class was an accident, and it was intended to be public all along. Adding the `public` keyword was the correct solution.

[1] I'm ignoring cv qualifiers here. You can put down your pitchforks.

Raymond Chen

**Follow**