Why can’t I find the injected name of a templated class’s templated base class?

devblogs.microsoft.com/oldnewthing/20240517-00

May 17, 2024

Raymond Chen

Some time ago, I wrote about how injected class names were the C++ feature you didn’t even realize that you were using. Injected class names let you use the plain name for the class being defined without needing to fully qualify it with namespaces and template parameters. Furthermore, injected class names are public and can be inherited.

“But wait, I’m trying to use the injected class name of my base class, but the compiler won’t accept it.”

```
template<typename T>
struct Base
{
    Base(T value);
};

template<typename T>
struct Derived : Base<T>
{
    Derived(T value) : Base(value) {}
};
```

This generates a compiler error.

```
// clang
error: member initializer 'Base' does not name a non-static data member or base class

    Derived(T value) : Base(value) {}
                       ^~~~~~~~~~

// gcc
error: class 'Derived<T>' does not have any field named 'Base'

    Derived(T value) : Base(value) {}
                       ^~~~

// msvc
error C2512: 'Base<T>': no appropriate default constructor available
error C2614: illegal member initialization: 'Base' is not a base or member
```

While it's true that `Base` is an injected type name of `Base`, it is also a *dependent type*, since its presence is indirectly dependent on the template type parameter `T`. Specifically, the name `Base` is provided by `Base<T>`, and that is dependent on `T`.

Now, you and I can plainly see that no matter how `Base<T>` is defined or specialized, there will always be a type named `Base` that refers to `Base<T>` due to name injection, but the compiler doesn't do this sort of fancy logical deduction.

Consider this alternate formulation:

```
template<typename T>
struct Base
{
    Base(int value);
    using Type = T;
};

template<typename T>
struct Derived : Base<T>
{
    Type m_value;
};
```

Here, `Type` is much more obviously a dependent type, since it is patently dependent on `T`. (And you might have a specialization of `Base<T>` which defines `Type` differently, or maybe doesn't even define it at all.)

At the time a template is parsed, non-dependent names are resolved immediately. The dependent names are resolved at the time a template is instantiated. This is known in the jargon as two-phase name lookup. Or as the LLVM blog once called it, the dreaded two-phase name lookup.

One solution is to eschew the injected name and use the original full name:

```
template<typename T>
struct Derived : Base<T>
{
    Derived(T value) : Base:<T>(value) {}
};
```

However, this can get clumsy if the base type name is unwieldy, say, because it comes from another namespace, or the template parameters are complex.

```
template<typename T>
struct Derived :
    Contoso::Faraway::Base<
        std::conditional_t<std::is_class_v<T>, T,
            std::tuple<T>>>
{
    Derived(T value) :
        Contoso::Faraway::Base<
            std::conditional_t<std::is_class_v<T>, T,
                std::tuple<T>>>(value) {}
};
```

But wait, all is not lost. We just need to postpone the lookup to the second phase by making it a dependent type. And there are two common ways of doing this.

- For data members and member functions, explicitly prefix them with `this->` to make them dependent on the full definition of the template class.
- For static members and member types, explicitly scope-qualify them with the name of the template class, and prefixing with `typename` if you're looking for a member type.

(If you want to access a static data member or static member function, both options are available, since you are allowed to use `this` to access a static data member or static member function.)

Therefore, one solution here is to write

```
template<typename T>
struct Derived : Base<T>
{
    Derived(T value) : Derived::Base(value) {}
};
```

Another is to explicitly import the type name `Base`:

```
template<typename T>
struct Derived : Base<T>
{
    using Base = typename Derived::Base;

    Derived(T value) : Base(value) {}
};
```

Additional reading: Why am I getting errors when my template-derived-class uses a member it inherits from its template-base-class?