In the example above, a `client` invoking `f` with an object of type `outer::U` fails to compile because `f(outer::U)` is declared in the nested `inner` namespace, which is not the same as declaring it in `outer`. Because ADL does not look into namespaces added with the **`using`** directive, ADL does not find the needed `outer::inner::f` function. Similarly, the type `V`, defined in namespace `outer::inner`, is not declared in the same namespace as the function `g` that operates on it. Hence, when `g` is invoked from within `client` on an object of type `outer::inner::V`, ADL again does not find the needed function `outer::g(outer::V)`.

Simply making the `inner` namespace **`inline`** solves both of these ADL-related problems. All transitively nested **`inline`** namespaces — up to and including the most proximate non-**`inline`** enclosing namespace — are treated as one with respect to ADL.

The ability to specialize templates declared in a nested `inline` namespace

The third property that distinguishes **`inline`** namespaces from conventional ones, even when followed by a **`using`** directive, is the ability to specialize a **`class template`** defined within an **`inline`** namespace from within an enclosing one; this ability holds transitively up to and including the most proximate non**`inline`** namespace:

```
namespace out                       // proximate noninline outer namespace
{
    inline namespace in1            // first-level nested inline namespace
    {
        inline namespace in2        // second-level nested inline namespace
        {
            template <typename T>   // primary class template general definition
            struct S { };

            template <>             // class template full specialization
            struct S<char> { };
        }

        template <>                 // class template full specialization
        struct S<short> { };
    }

    template <>                     // class template full specialization
    struct S<int> { };
}

using namespace out;                // conventional using directive

template <>
struct S<int> { };                  // Error, cannot specialize from this scope
```

Note that the conventional nested namespace `out` followed by a **`using`** directive in the enclosing namespace does not admit specialization from that outermost namespace, whereas