Finally, declaring a template in the nested **inline** namespace `inner` in the example above and then subsequently defining it in the enclosing `outer` namespace has the same effect of making declared symbols ambiguous in the `outer` namespace:

```
namespace outer                                          // enclosing namespace
{                                                        // BAD IDEA
    inline namespace inner                               // nested namespace
    {
        template<typename T> struct S3;                  // declarations of
        template<typename T> void f3();                  // various class
        template<typename T> void g3(T v);               // and function
        struct A3 { template <typename T> void h3(); };  // templates
    }

    template<typename T> struct S3 { };                  // definitions of
    template<typename T> void f3() { }                   // unrelated class
    template<typename T> void g3(T v) { }                // and function
    template<typename T> void A3::h3() { }               // templates

    template<> struct S3<int> { };     // Error, S3 is ambiguous in outer.
    template<> void f3<int>() { }      // Error, f3 is ambiguous in outer.
    void g3(int) { }                   // OK, g3 is an overload definition.
    template<> void A3::h3<int>() { }  // Error, h2 is ambiguous in outer.
}
```

Note that~~, although the definition for a member function template must be located directly within the namespace in which it is declared~~, a class or function template, once declared, may instead be defined in a different scope by using an appropriate name qualification:

```
template <typename T> struct outer::S3 { };        // OK, enclosing namespace
template <typename T> void outer::inner::f3() { }  // OK, nested namespace
template <typename T> void outer::g3(T v) { }      // OK, enclosing namespace
template <typename T> void outer::A3::h3<T>() { }  // Error, ill-formed

namespace outer
{
    inline namespace inner
    {
        template <typename T> void A3::h3() { }    // OK, within same namespace
    }
}
```

Also note that, as ever, the corresponding definition of the declared template must have been seen before it can be used in a context requiring a complete type. The importance of ensuring that all specializations of a template have been seen before it is used substantively (i.e., ODR-used) cannot be overstated, giving rise to the only limerick, which is actually part of the normative text, in the C++ Language Standard[10]:

[10] See **iso11a**, section 14.7.3, "Explicit specialization," paragraph 7, pp. 375–376, specifically p. 376.