`inline namespace std {}` before including any standard headers. This practice is, however, explicitly called out as ill-formed within the C++11 Standard. Although not uniformly diagnosed as an error by all compilers, attempting this forbidden practice is apt to lead to surprising problems even if not diagnosed as an error immediately.

Inconsistent use of `inline` keyword is ill formed, no diagnostic required

It is an ODR violation, IFNDR, for a nested namespace to be `inline` in one translation unit and non`inline` in another. And yet, the motivating use case of this feature relies on the linker to actively complain whenever different, incompatible versions — nested within different, possibly `inline`-inconsistent, namespaces of an ABI — are used within a single executable. Because declaring a nested namespace `inline` does not, by design, affect linker-level symbols, developers must take appropriate care, such as effective use of header files, to defend against such preventable inconsistencies.

Annoyances

Inability to redeclare across namespaces impedes code factoring

An essential feature of an `inline` namespace is the ability to declare a template within a nested `inline` namespace and then specialize it within its enclosing namespace. For example, we can declare

- a type template, `S0`
- a couple of function templates, `f0` and `g0`
- and a member function template `h0`, which is similar to `f0`

in an `inline` namespace, `inner`, and specialize each of them, such as for `int`, in the enclosing namespace, `outer`:

```
namespace outer                                        // enclosing namespace
{
    inline namespace inner                             // nested namespace
    {
        template<typename T> struct S0;                // declarations of
        template<typename T> void f0();                // various class
        template<typename T> void g0(T v);             // and function
        struct A0 { template <typename T> void h0(); };  // templates
    }

    template<> struct S0<int> { };                     // specializations
    template<> void f0<int>() { }                      // of the various
    void g0(int) { }  /* overload not specialization */  // class and function
    template<> void A0::h0<int>() { }                  // declarations above
}                                                      // in outer namespace
```