```
    struct ThingAggregate
    {
        // ...
        VersionedThing d_thing;
        // ...
    };
}
```

This new `ThingAggregate` type does not have the versioned **`inline`** namespace as part of its mangled name; it does, however, have a completely different layout if built with `MY_THING_VERSION_1` defined versus `MY_THING_VERSION_2` defined. Linking a program with mixed versions of these flags will result in runtime failures that are decidedly difficult to diagnose.

This same sort of problem will arise for functions taking arguments of such types; calling a function from code that is wrong about the layout of a particular type will result in stack corruption and other undefined and unpredictable behavior. This macro-induced problem will also arise in cases where an old object file is linked against new code that changes which namespace is **`inline`**d but still provides the definitions for the old version namespace. The old object file for the client can still link, but new object files using the headers for the old objects might attempt to manipulate those objects using the new namespace.

The only viable workaround for this approach is to propagate the **`inline`** namespace hierarchy through the entire software stack. Every object or function that uses `my::VersionedThing` needs to also be in a namespace that differs based on the same control macro. In the case of `ThingAggregate`, one could just use the same `my::v1` and `my::v2` namespaces, but higher-level libraries would need their own `my`-specific nested namespaces. Even worse, for higher-level libraries, every lower-level library having a versioning scheme of this nature would need to be considered, resulting in having to provide the full cross-product of nested namespaces to get link-time protection against mixed-mode builds.

This need for layers above a library to be aware of and to integrate into their own structure the same namespaces the library has removes all or most of the benefits of using **`inline`** namespaces for versioning. For an authentic real-world case study of heroic industrial use — and eventual disuse — of **`inline`**-namespaces for versioning, see *Appendix — Case study of using **`inline`** namespaces for versioning* on page 1083.

Relying on `inline` namespaces to solve library evolution

Inline namespaces might be misperceived as a complete solution for the owner of a library to evolve its API. As an especially relevant example, consider the C++ Standard Library, which itself does not use inline namespaces for versioning. Instead, to allow for its anticipated essential evolution, the Standard Library imposes certain special restrictions on what is permitted to occur within its own `std` namespace by dint of deeming certain problematic uses as either ill formed or otherwise engendering **undefined behavior**.