```
// my_thing.cpp:
#include <my_thing.h>

namespace my                      // outer namespace (used directly by clients)
{
    inline namespace impl_v2      // inner namespace (for implementer use only)
    {
        Thing::Thing() : d(0.0)   // Load 8-byte value into Thing's data member.
        {
        }
    }
}
```

Now clients that attempt to link against the new library will not find the linker symbol, such as `_Z...impl_v1...v`, and the link stage will fail. Once clients recompile, however, the undefined linker symbol will match the one available in the new `my_thing.o`, such as `_Z...impl_v2...v`, the link stage will succeed, and the program will again work as expected. What's more, we have the option of keeping the original implementation. In that case, existing clients that have not as yet recompiled will continue to link against the old version until it is eventually removed after some suitable deprecation period.

As a more realistic second example of using **inline** namespaces to guard against linking incompatible versions, suppose we have two versions of a `Key` class in a security library in the enclosing namespace, `auth` — the original version in a regular nested namespace `v1`, and the new current version in an **inline** nested namespace `v2`:

```
#include <cstdint>  // std::uint32_t, std::unit64_t

namespace auth      // outer namespace (used directly by clients)
{
    namespace v1    // inner namespace (optionally used by clients)
    {
        class Key
        {
        private:
            std::uint32_t d_key;
                // sizeof(Key) is 4 bytes.

        public:
            std::uint32_t key() const;  // stable interface function

            // ...
        };
    }

    inline namespace v2    // inner namespace (default current version)
    {
        class Key
```