that require specialization or operator-like functions, such as swap, defined for local types within those nested namespaces. In those cases, **inline** namespaces would be required to preserve the desired as-if properties.

Even without either of these two needs, another property of an **inline** namespace differentiates it from a non**inline** one followed by a **using** directive. Recall from *Description — Loss of access to duplicate names in enclosing namespace* on page 1056 that a name in an outer namespace will hide a duplicate name imported via a **using** directive, whereas any access to that duplicate name within the enclosing namespace would be ambiguous when that symbol is installed by way of an **inline** namespace. To see why this more forceful clobbering behavior might be preferred over hiding, suppose we have a communal namespace abc that is shared across multiple disparate headers. The first header, abc_header1.h, represents a collection of logically related small functions declared directly in abc:

```
// abc_header1.h:
namespace abc
{
    int i();
    int am();
    int smart();
}
```

A second header, abc_header2.h, creates a suite of many functions having tiny function names. In a perhaps misguided effort to avoid clobbering other symbols within the abc namespace having the same name, all of these tiny functions are sequestered within a nested namespace:

```
// abc_header2.h:
namespace abc
{
    namespace nested  // Should this namespace have been inline instead?
    {
        int a();  // lots of functions with tiny names
        int b();
        int c();
        // ...
        int h();
        int i();  // might collide with another name declared in abc
        // ...
        int z();
    }

    using namespace nested;  // becomes superfluous if nested is made inline
}
```

Now suppose that a client application includes both of these headers to accomplish some task: