best; see *Potential Pitfalls — Relying on* ***inline*** *namespaces to solve library evolution* on page 1077.

Providing separate namespaces for each successive version has an additional advantage in an entirely separate dimension: avoiding inadvertent, difficult-to-diagnose, latent **linkage** defects. Though not demonstrated by this specific example, cases do arise where simply changing which of the version namespaces is declared ***inline*** might lead to an **ill formed, no-diagnostic required** (IFNDR) program. This issue might ensue when one or more of its **translation units** that use the library are not recompiled before the program is relinked to the new static or dynamic library containing the updated version of the library software; see *Link-safe ABI versioning* below.

For distinct nested namespaces to guard effectively against accidental link-time errors, the symbols involved have to (1) reside in object code (e.g., a **header-only library** would fail this requirement) and (2) have the same **name mangling** (i.e., linker symbol) in both versions. In this particular instance, however, the signature of the `parse` **member function** of `parser` did change, and its mangled `name` will consequently change as well; hence the same `undefined symbol` link error would result either way.

Link-safe ABI versioning

inline namespaces are not intended as a mechanism for source-code versioning; instead, they prevent programs from being **ill formed** due to linking some version of a library with client code compiled using some other, typically older version of the same library. Below, we present two examples: a simple pedagogical example to illustrate the principle followed by a more real-world example. Suppose we have a library **component** `my_thing` that implements an example type, `Thing`, which wraps an ***int*** and initializes it with some **value** in its **default constructor** defined out-of-line in the `cpp` file:

```
struct Thing  // version 1 of class Thing
{
    int i;    // integer data member (size is 4)
    Thing();  // original noninline constructor (defined in .cpp file)
};
```

Compiling a source file with this version of the header included might produce an object file that can be incompatible yet linkable with an object file resulting from compiling a different source file with a different version of this header included:

```
struct Thing   // version 2 of class Thing
{
    double d;  // double-precision floating-point data member (size is 8)
    Thing();   // updated noninline constructor (defined in .cpp file)
};
```

To make the problem that we are illustrating concrete, let's represent the client as a `main` program that does nothing but create a `Thing` and print the `value` of its only **data member**, `i`.