```
}

#endif
```

The implementation file `my_thing.cpp` contains all of the non**`inline`** function bodies that will be translated separately into the `my_thing.o` file:

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

namespace my                  // outer namespace (used directly by clients)
{
    inline namespace impl_v1  // inner namespace (for implementer use only)
    {
        Thing::Thing() : i(0)  // Load a 4-byte value into Thing's data member.
        {
        }
    }
}
```

Observing common good practice, we include the header file of the component as the first substantive line of code to ensure that — irrespective of anything else — the header always compiles in isolation, thereby avoiding insidious include-order dependencies.[5] When we compile the source file `my_thing.cpp`, we produce an object file `my_thing.o` containing the definition of the same linker symbol, such as `_ZN2my7impl_v15ThingC1Ev`, for the default constructor of `my::Thing` needed by the client:

```
$ g++ -c my_thing.cpp
```

We can then link `main.o` and `my_thing.o` into an executable and run it:

```
$ g++ -o prog main.o my_thing.o
$ ./prog

0
```

Now, suppose we were to change the definition of `my::Thing` to hold a **`double`** instead of an **`int`**, recompile `my_thing.cpp`, and then relink with the original `main.o` without recompiling `main.cpp` first. None of the relevant linker symbols would change, and the code would recompile and link just fine, but the resulting binary `prog` would be IFNDR: the client would be trying to print a 4-byte, **`int`** data member, `i`, in `main.o` that was loaded by the library component as an 8-byte, **`double`** into `d` in `my_thing.o`. We can resolve this problem by changing — or, if we didn't think of it in advance, by adding — a new **`inline`** namespace and making that change there:

[5]See **lakos20**, section 1.6.1, "Component Property 1," pp. 210–212.