all of the **inline** namespaces do. Function templates behave similarly except that — unlike class templates, whose ~~definitions~~ must reside ~~entirely~~ within the namespace ~~in which they are declared~~ — a function template can be *declared* within a nested namespace and then be *defined* from anywhere via a **qualified name**:

```
namespace out                       // proximate noninline outer namespace
{
    inline namespace in1            // first-level nested inline namespace
    {
        template <typename T>       // function template declaration
        void f();

        template <>                 // function template (full) specialization
        void f<short>() { }
    }

    template <>                     // function template (full) specialization
    void f<int>() { }
}

template <typename T>               // function template general definition
void out::in1::f() { }
```

An important takeaway from the examples above is that every template entity — be it class or function — *must* be declared in *exactly* one place within the collection of namespaces that comprise the **inline** namespace set. In particular, declaring a class template in a nested **inline** namespace and then subsequently defining it in a containing namespace is not possible because, unlike a function definition, a type definition cannot be placed into a namespace via name qualification alone:

```
namespace outer
{
    inline namespace inner
    {
        template <typename T>       // class template declaration
        struct Z;                   // (if defined, must be within same namespace)

        template <>                 // class template full specialization
        struct Z<float> { };
    }

    template <typename T>           // inconsistent declaration (and definition)
    struct Z { };                   // Z is now ambiguous in namespace outer.

    const int i = sizeof(Z<int>);   // Error, reference to Z is ambiguous.

    template <>                     // attempted class template full specialization
```