Note that, in the case of `g0` in this example, the "specialization" `void g0(int)` is a non-template *overload* of the function template `g0` rather than a specialization of it. We *cannot*, however, portably[9] declare these templates within the `outer` namespace and then specialize them within the `inner` one, even though the `inner` namespace is `inline`:

```
namespace outer                                     // enclosing namespace
{
    template<typename T> struct S1;                 // class template
    template<typename T> void f1();                 // function template
    template<typename T> void g1(T v);              // function template

    struct A1 { template <typename T> void h1(); }; // member function template

    inline namespace inner                          // nested namespace
    {                                               // BAD IDEA
        template<> struct S1<int> { };              // Error, S1 not a template
        template<> void f1<int>() { }               // Error, f1 not a template
        void g1(int) { }                            // OK, overloaded function
        template<> void A1::h1<int>() { }           // Error, h1 not a template
    }
}
```

Attempting to declare a template in the `outer` namespace and then define, effectively redeclaring, it in an `inline` inner one causes the name to be inaccessible within the `outer` namespace:

```
namespace outer                                     // enclosing namespace
{                                                   // BAD IDEA
    template<typename T> struct S2;                 // declarations of
    template<typename T> void f2();                 // various class and
    template<typename T> void g2(T v);              // function templates

    inline namespace inner                          // nested namespace
    {
        template<typename T> struct S2 { };         // definitions of
        template<typename T> void f2() { }          // unrelated class and
        template<typename T> void g2(T v) { }       // function templates
    }

    template<> struct S2<int> { };     // Error, S2 is ambiguous in outer.
    template<> void f2<int>() { }      // Error, f2 is ambiguous in outer.
    void g2(int) { }                   // OK, g2 is an overload definition.
}
```

[9] GCC provides the `-fpermissive` flag, which allows the example containing specializations within the inner namespace to compile with warnings. Note again that `g1(int)`, being an *overload* and not a *specialization*, wasn't an error and, therefore, isn't a warning either.