Unless both type and variable entities are declared within the same scope, no preference is given to variable names; the name of an entity in an inner scope hides a like-named entity in an enclosing scope:

```
void f()
{
    double B;                 static_assert(sizeof(B) == 8, "");  // variable
    {                         static_assert(sizeof(B) == 8, "");  // variable
        struct B { int d; };  static_assert(sizeof(B) == 4, "");  // type
    }                         static_assert(sizeof(B) == 8, "");  // variable
}
```

When an entity is declared in an enclosing **namespace** and another entity having the same name hides it in a *lexically* nested scope, then (apart from **inline** namespaces) access to a hidden element can generally be recovered by using scope resolution:

```
struct C { double d; };  static_assert(sizeof(  C) == 8, "");

void g()
{                        static_assert(sizeof(  C) == 8, "");  // type
    int C;               static_assert(sizeof(  C) == 4, "");  // variable
                         static_assert(sizeof(::C) == 8, "");  // type
}                        static_assert(sizeof(  C) == 8, "");  // type
```

A conventional nested namespace behaves as one might expect:

```
namespace outer
{
    struct D { double d; }; static_assert(sizeof(       D) == 8, ""); // type

    namespace inner
    {                       static_assert(sizeof(       D) == 8, ""); // type
        int D;              static_assert(sizeof(       D) == 4, ""); // var
    }                       static_assert(sizeof(       D) == 8, ""); // type
                            static_assert(sizeof(inner::D) == 4, ""); // var
                            static_assert(sizeof(outer::D) == 8, ""); // type
    using namespace inner;//static_assert(sizeof(       D) == 0, ""); // Error
                            static_assert(sizeof(inner::D) == 4, ""); // var
                            static_assert(sizeof(outer::D) == 8, ""); // type
}                           static_assert(sizeof(outer::D) == 8, ""); // type
```

In the example above, the inner variable name, `D`, hides the outer type with the same name, starting from the point of `D`'s **declaration** in `inner` until `inner` is closed, after which the unqualified name `D` reverts to the type in the `outer` namespace. Then, right after the subsequent **`using namespace`** `inner;` directive, the meaning of the unqualified name `D` in `outer` becomes ambiguous, shown here with a **`static_assert`** that is commented out; any attempt to refer to an unqualified `D` from here to the end of the scope of `outer` will fail to compile. The type entity declared as `D` in the `outer` namespace can, however, still be