Let's now suppose it *is* our intention to suppress generation of the default constructor, and to make our intention clear, we elect to explicitly declare and **delete** it:

```
struct S2  // Implicit declaration of the default constructor is suppressed.
{
    S2() = delete;  // explicit declaration of ~~inaccessible~~ default constructor
    S2(int);        // explicit declaration of value constructor
};

S2 y2(5);  // OK, invokes the explicitly declared value constructor
S2 x2;     // Error, use of deleted function, S2::S2()
```

By declaring and then deleting the default constructor we have, it would appear that we (1) made our intentions clear and (2) improved diagnostics for our clients at the cost of a single extra line of self-documenting code. Ah, if only C++ were that straightforward.

Deleting certain special member functions — i.e., *default* constructor, *move* constructor, or *move*-assignment operator — that are not necessarily implicitly declared can have non-obvious consequence that adversely affect subtle compile-time properties of a class. One such subtle property is whether the compiler considers it to be a **literal type**, i.e., a type whose *value* is eligible for use as part of a **constant expression**. This same property of being a literal type is what determines whether an arbitrary type may be passed by value in the interface of a **constexpr** function; see Section 2.1."**constexpr** Functions" on page 257.

As a simple illustration of a subtle compile-time difference between S1 and S2, consider this practically useful *pattern* for a developer's "test" function that will compile if and only if its by-value parameter, x, is of a literal type:

```
constexpr int test(S0 x) { return 0; }  // OK,    S0 is     a literal type.
constexpr int test(S1 x) { return 0; }  // Error, S1 is not a literal type.
constexpr int test(S2 x) { return 0; }  // OK,    S2 is     a literal type.
```

For the compiler to treat a given class type as a literal type, it must, among other things, have at least one constructor (other than the *copy* or *move* constructor) declared as **constexpr**.

In the case of the empty S0 class, the implicitly generated default constructor is **trivial** and so it is implicitly *declared* **constexpr** too. Class S1's explicitly declared *non***constexpr** value constructor suppresses the declaration of its only **constexpr** constructor, the default constructor; hence, S1 does not qualify as a *literal type*.

Finally, by conspicuously declaring and deleting S2's default constructor, we *declare* it nonetheless. What's more, the declaration brought about by deleting it is the same as if it had been generated implicitly (or declared explicitly and then defaulted); hence, S2, unlike S1, *is* a literal type. Go figure!