- `a8` — Because `a8` has static storage duration, it is first *zero* initialized, i.e., `a8.i` is set to 0, and then it is *default* initialized, which is a no-op for the same reasons that it is a no-op for `a1`.

Finally, note that a scalar can be thought of as if it were an array of one element, though note that scalars are never subject to **array-to-pointer decay**; in fact, if we were to take the address of any scalar and add `1` to it, the new pointer value would represent the one-past-the-end iterator for that scalar's implied array of length `1`. Similarly, scalars can be initialized using aggregate initialization, just as if they were single-element arrays, where the braced list for a scalar can contain zero or one elements. In C++03, however, scalars cannot be initialized from an empty brace:

```
int    i = { };       // Error in C++03; OK in C++11 (i is 0).
int    j = { 1 };     // OK, ~~i~~ is 1.
double k = { 3.14 };  // OK, k is 3.14.
```

Braced initialization in C++11

Everything we've discussed so far, including braced initialization of aggregates, is well defined in C++03. This same braced-initialization syntax — modified slightly so as to preclude narrowing conversions (see the next section) — is extended in C++11 to work consistently and uniformly in many new situations. This enhanced braced-initialization syntax is designed to better support the two dual initialization categories discussed in *C++03 initialization syntax review* on page 215 as well as entirely new capabilities including language-level support for lists of initial values implemented using the C++ Standard Library's `std::initializer_list` class template.

C++11 restrictions on narrowing conversions

Narrowing conversions, a.k.a. **lossy conversions**, are a notorious source of runtime errors. One of the important properties of list initializations implemented using the C++11 braced-initialization syntax is that error-prone narrowing conversions are no longer permitted. Consider, for example, an `int` array, `ai`, initialized with various built-in *literal* values:

```
int ai[] =
{         //     C++03   C++11
    5,    // (0) OK      OK
    5.0,  // (1) OK      Error, narrowing double to int conversion is not allowed.
    5.5,  // (2) OK      Error, narrowing double to int conversion is not allowed.
    "5",  // (3) Error   Error, no const char* to int conversion exists.
};
```

In C++03, floating-point literals would be coerced to fit within an integer even if the conversion was known to be lossy, e.g., line (2) in the code snippet above would initialize `ai[2]` to `5`. By contrast, C++11 disallows *any* such implicit conversions in braced initializations even when the conversion is known *not* to be lossy, e.g., element `ai[1]` above.