```
struct I1b { int i;  constexpr I1b()       { } };  // Error, i is not init

struct I2a { int i;            I2a() = default; };  // OK, but not constexpr
struct I2b { int i;  constexpr I2b() = default; };  // Error, i is not init

struct I3a { int i;            I3a() : i(0) { } };  // OK, i is init
struct I3b { int i;  constexpr I3b() : i(0) { } };  // OK, literal type

struct S0  { I3b v; /* implicit default ctor */ };  // OK, literal type

struct S1a { I3b v;            S1a()       { } };  // OK, v is init
struct S1b { I3b v;  constexpr S1b()       { } };  // OK, literal type

struct S2a { I3b v;            S2a() = default; };  // OK, literal type
struct S2b { I3b v;  constexpr S2b() = default; };  // OK, literal type
```

The example code above illustrates the subtle differences between a data member of *scalar* literal type, e.g., **int** and one of *user-defined* literal type, e.g., `I3b`. Unlike `I1a`, which leaves its own data member, `i`, uninitialized, `S1a` invariably zero-initializes its `i`. Therefore, attempting to apply **constexpr** to the constructor of `I1a` is ill formed, which is not the case for `S1a`, as illustrated by `I1b` and `S1b` above.

Note that, although every literal type needs to have a way to be constructed in a context requiring a constant expression, not *every* constructor of a literal type needs to be **constexpr**; see *Literal types defined* on page 278.

6. **Value initialization** and **aggregate initialization**, although not always resulting in constructor invocation, can still occur at compile time. These kinds of initialization must involve only those operations that can occur during constant evaluation.

   For types having a user-provided default constructor, value initialization implies invoking that constructor, thus requiring it to be declared **constexpr** for it to be evaluated as part of a constant expression. For types having an implicitly defined (or defaulted; see Section 1.1."Defaulted Functions" on page 33) default constructor, value initialization will first zero-initialize all base-class objects and members and will then **default-initialize** the object itself, which places similar restriction on the constructor being **constexpr**. If, however, an implicitly defined or defaulted constructor is also **trivial**, its invocation will be skipped.[6] A default constructor is trivial if (a) it is implicitly

[6]The original intent was to enable any initialization that involved only those operations that could be evaluated at compile time to be a valid initialization for a literal type. That a trivial default constructor was insufficient to make a class a literal type, as it was not going to be a **constexpr** constructor, was a flaw originally noted by Alisdair Meredith; see CWG issue 644 (**meredith07**). The resolution for this issue was inadvertently undone before C++11 shipped by mistakenly allowing aggregate initialization in lieu of trivial initialization with other resolutions; see CWG issues 981 (**dosreis09**) and 1071 (**krugler10a**). This flaw was identified again (CWG issue 1452; **smith11b**), and all relevant compilers adopted the proposed resolution as a fix, but the Standard itself did not. C++20 removes the requirement that all members and base classes be initialized in a **constexpr** constructor (**johnson19**), removing the flaw by making trivial default constructors **constexpr**.