- The requirement to have at least one **constexpr** constructor that is not a *copy* or *move* constructor is just that: to have at least one. There is no requirement that such a constructor be invocable at compile time, e.g., it could be declared **private**, or even that it be defined; in fact, a deleted constructor (see Section 1.1."Deleted Functions" on page 53) satisfies the requirement:

```
struct UselessLiteralType
{
    constexpr UselessLiteralType() = delete;
};
```

- Many uses of literal types in **constexpr** functions will require additional **constexpr** functions to be defined (not merely declared), such as a *move* or *copy* constructor:

```
struct Lt  // literal type having nonconstexpr copy constructor
{
    constexpr Lt(int c) { }  // valid constexpr value constructor
    Lt(const Lt& ) { }       // nonconstexpr copy constructor
};

constexpr int processByValue(Lt t) { return 0; }  // valid constexpr function

static_assert(processByValue(Lt(7)) == 0, "");
    // Error, but might work on some platforms due to elided copy

constexpr Lt s{7};  // braced-initialized object of type Lt

static_assert(processByValue(s) == 0, "");  // Error, nonconstexpr copy ctor
```

In the code example above, we have a literal type, `Lt`, for which we have explicitly declared a non**constexpr** copy constructor. We then defined a valid **constexpr** function, `processByValue`, taking an `Lt` (by value) as its only argument. Invoking the function by constructing an object of `Lt` from a literal **int** value enables the compiler to elide the copy. Platforms where the copy is elided might allow this evaluation at compile time, while on other platforms there will be an error. When we consider using an independently constructed **constexpr** variable (i.e., `s`), the `copy` can no longer be elided, and since the copy constructor is declared explicitly to be non**constexpr**, the compile-time assertion fails to compile on all platforms; see Section 2.1."**constexpr** Variables" on page 302.

- Although a pointer or reference is always (by definition) a *literal type*, if the type being pointed to is not itself a literal type, then the referenced object cannot be used during constant expression evaluation.