3. Any variable declared **constexpr** must be of literal type; all literal types are, among other things, trivially destructible:

```
struct Lt  // literal type
{
    constexpr Lt() { }  // constexpr constructor
    ~Lt() = default;    // default trivial destructor
};

constexpr Lt lt;  // OK, Lt is a literal type.

struct Nlt  // nonliteral type
{
    Nlt()  { }  // cannot initialize at compile time
    ~Nlt() { }  // cannot skip non-trivial destruction
};

constexpr Nlt nlt;  // Error, Nlt is not a literal type.
```

   Since all literal types are trivially destructible, the compiler does not need to emit any special code to manage the end of the lifetime of a **constexpr** variable, which can essentially live "forever" — i.e., until the program exits.[2]

4. Unlike integral constants, non**static** data members cannot be **constexpr**. Only variables at global or **namespace** scope, automatic variables, or **static** data members of a **class** or **struct** may be declared **constexpr**. Consequently, any given **constexpr** variable is a top-level object, never a subobject of another, possibly non-**constexpr**, object:

```
               constexpr int i = 17;    // OK, file scope
namespace ns { constexpr int j = 34; }  // OK, namespace scope

struct C
{
    static constexpr int k = 51;  // OK, static data member
           constexpr int l = 68;  // Error, constexpr nonstatic data member
};

void g()
{
    static constexpr int m = 85;  // OK
           constexpr int n = 92;  // OK
}
```

   Recall, however, that nonstatic data members of **constexpr** objects are implicitly **constexpr** and therefore can be used directly in any constant expressions:

[2]In C++20, literal types can have non-trivial destructors, and the destructors for **constexpr** variables will be invoked under the same conditions that a destructor would be invoked for a non**constexpr** global or **static** variable.