same location in memory, an attempt to access the second data member when the initial member sequence is other than an exact match nonetheless has undefined behavior. Moreover, writing to any part of any other, noncompatible union member, irrespective of its relative **physical** position in its POD-**struct** (e.g., writing to `aa.h`), renders the previously active member of the union (e.g., `bb`) inactive, thereby precluding access to *any* members of the original pair of partially compatible POD-**structs** (e.g., `bb` and `cc`). On most platforms, however, the buggy code in the previous example is likely to compile and perform as though there was no undefined behavior; see *Potential Pitfalls — Misuse of* `unions` on page 505.

3. **Lifetime of an object begins at allocation** — Scalar types are trivial types, which means no code need run to either construct or destroy scalar objects. POD-**struct** types, like the scalars they comprise, also are **trivial**. The lifetime of a POD object starts when memory is first acquired for it, such as by a **variable** declaration or a call to the **new operator**. However, starting the lifetime of a POD does not guarantee that it has been initialized. Consider the case of declaring an **int** as a local variable:

```
void test2()
{
    int  x;       // x is not initialized, but lifetime begins.
    int* p = &x;  // We cannot read x but can take its address.
    int  y = x;   // Bug, read of uninitialized x
    *p = 5;       // We can write to x, thereby initializing it.
    int  z = x;   // Now we can read x.
}
```

Similarly, the lifetime of a POD object ends when its memory is reclaimed, such as by going out of scope, or when it is repurposed by constructing a new object in that memory; the **destructor** of a POD is always trivial, and nothing will execute when a POD object is destroyed. Note that explicit invocation of a **trivial destructor** will not end the lifetime of an object, including a POD object.[3]

Although a POD can be declared **const** or contain a non**static** **const** data member, such a POD cannot exist in an uninitialized state; attempting to create an object that requires **const** data member initialization will fail to compile if the initializer is omitted:

```
struct S2          // POD type containing two scalar data members
{
    const int* p;  // Pointers, but not references, can be POD data members.
    const int  i;  // Note that const data members must be initialized.
};

S2 s2a;            // Error, uninitialized const data member, s2a.i
S2 s2b = { 0, 5 }; // OK, const data member s2b.i is initialized to 5.
```

[3] As of C++20, running the destructor of any object — even a POD — ends its lifetime, and assigning a value to it after the fact would have undefined behavior; see CWG issue 2256 (**smith16b**).