we can return to our original implementation of `readAndProcess` eschewing the cleanup code while retaining program correctness.

We might refer to a class like `Point3`, which validates its invariants on destruction, as **notionally trivially destructible** because it can be used *as if* it were trivially destructible. In generic software that does not know that a type is notionally trivially destructible, the `Point3` class might suffer some performance loss relative to `Point`, especially in a debug build, but the semantics of a *correct* program do not change. Note, however, that when we skip `Point3`'s destructor invocation, we give up — even in debug mode — the **defensive checks** in `Point3`'s destructor that might catch a bug in our program.

Though useful for human discourse, notionally trivially destructible types are not considered trivially destructible by the compiler or by any general-purpose library and thus are neither literal types (see Section 2.1."**constexpr** Functions" on page 257) nor **trivially copyable types**, as both of these properties require **trivial destructibility**. An notionally trivially destructible type cannot, therefore, be used where either of these properties is an arbiter of correctness:

```
#include <cstring>  // std::memcpy

char array1[Point {1, 2}.d_x];  // OK,    Point  is   a literal type.
char array2[Point3{1, 2}.d_x];  // Error, Point3 isn't a literal type.

void f(Point*  d, const Point*  s) { std::memcpy(d, s, sizeof *s); }  // OK
void f(Point3* d, const Point3* s) { std::memcpy(d, s, sizeof *s); }  // Bug, UB
    // Point3 is not trivially copyable; hence, f's behavior is undefined (UB).
```

In the code snippet above, using `Point3` in an array-size computation will fail to compile, whereas the original `Point` class will work just fine; see *Compile-time constructible, literal types (trivially destructible)* on page 462. Although using `std::memcpy` to copy objects of trivially copyable type such as `Point` is valid (see *Fixed-capacity string (trivially copyable)* on page 470), using `std::memcpy` to propagate values of a *non*-trivially copyable type such as `Point3` has undefined behavior; see *Potential Pitfalls — Ineligible use of `std::memcpy`* on page 497.

In a more aggressive version of this runtime optimization technique, even types that allocate memory can be considered notionally trivially destructible when the memory that would be deallocated by the destructor can somehow be reclaimed in other ways.[27]

Finally, since the code in the destructor for `Point3` is active *only* in a **debug build**, we might be tempted to define a point class (e.g., `Point4`) for which the entire user-provided destructor

[27]The introduction of `std::pmr::monotoni_resource` and `std::pmr::unsynchronized_pool_resource` in C++17 enables omitting destructor invocation for some *non*-trivially destructible types through the use of local allocators supplied at construction that reclaim all associated memory when they are destroyed, independently of whether the objects that requested the memory ever freed the memory themselves; see **lakos17a**, time 00:38:19. Note that this optimization technique can also be applied at the design level — e.g., to implement efficient garbage collection of cyclically connected networks of objects allocated from a single local memory arena; see **lakos19**, time 01:12:45.