```
            shapeEncodings[elemIdx++].d_vertex.~Point2();
        }
    }
}  // The local shapeEncodings array goes out of scope.
```

But what if we know that our point class doesn’t manage any resources that might leak? A common practice during software testing, and sometimes even in production, is to check the invariants of a class within its destructor to verify that no class operation or spurious program defect has left the object in an invalid state. Imagine applying this technique to a variation on the original `Point` class for which `d_x` and `d_y` are always within the range -5000 to +5000. We might choose to instrument our revised class (e.g., `Point3`) to enforce these invariants during development:

```
#include <cassert>  // standard C assert macro

struct Point3  // trivially constructible but not trivially destructible
{
    int d_x, d_y;  // same data as before

    ~Point3()      // Destructor is user-provided; hence, non-trivial.
    {
        assert(-5000 <= d_x);  assert(d_x <= 5000);
        assert(-5000 <= d_y);  assert(d_y <= 5000);
    }
};
```

Class `Point3` checks that both `d_x` and `d_y` satisfy their object invariants during destruction, but only in a *debug* build — i.e., one in which the `NDEBUG` macro is *not* defined.[26] The addition of this user-provided destructor again makes our point class *non*-trivially destructible in *any* build mode. Just as for `ShapeElem2`, we must provide a destructor for a **`union`** element (e.g., `ShapeElem3`) employing `Point3` as the type of its `d_vertex` member:

```
union ShapeElem3  // like ShapeElem except no longer trivially destructible
{
    int    d_numVertices;
    Point3 d_vertex;   // revised point having a non-trivial destructor

    ~ShapeElem3() { }  // required since Point3's non-trivially destructible
};
```

Again, `ShapeElem3`’s empty destructor does not invoke the destructor for either of its members, but, unlike `ShapeElem2`, failing to destroy a possibly active `Point3` member is acceptable because `Point3` has a destructor that neither releases a resource nor produces a side effect that would — in *any* way — affect the correctness of an already-correct program. Thus,

[26] A proposal for a more general C++ assertion facility — known widely as “contracts” — narrowly missed being included in C++20 and is the focus of an ongoing study group (SG21) for future inclusion in C++; see **dosreis18**.