The Order That Was in Two Places

A post-mortem on nano_match: a use-after-free that never crashed, and what an object pool takes away from you in exchange for flat tail latency.

Abir Deol · University of Alberta, Computing Science · July 2026

Abstract. nano_match is a limit order book and matching engine in C++ with price-time priority and no dynamic allocation on the matching path. Order objects come from a fixed-size pool and are recycled. This is a post-mortem on a use-after-free that never crashed: a cancelled order was returned to the pool while a price level still held a pointer to it, the pool handed that same memory to the next incoming order, and the book quietly began matching at prices that were not real. The engine kept running and kept producing fills. That was the problem.

1. The system

A limit order book is a set of price levels, each holding a queue of resting orders in arrival sequence. Price-time priority means the best price matches first, and within a price, the order that arrived first matches first. The queue ordering is not cosmetic. It is the guarantee the venue makes to participants.

Because the matching path must not touch the system allocator, order objects are preallocated in a pool and handed out by pointer. Price levels hold intrusive linked lists: the next and previous pointers live inside the order object. When an order is fully filled or cancelled it is released back to the pool for reuse. No malloc, no free, no page faults, and consequently no allocator variance in the tail latency.

An object pool is a lifetime contract. The pool guarantees the memory stays valid. It guarantees nothing at all about whether anyone is still pointing at it.

2. The symptom

The book crossed. A bid appeared at a price above the best ask, which in a correct engine cannot survive a single matching pass, because any such pair matches immediately by construction. Alongside that, fills came out in the wrong sequence within a price level: an order that arrived later matched ahead of one that had been resting longer.

There was no crash, no assertion, no allocator complaint. The pool memory was valid, properly aligned, fully initialized, and owned by the engine the entire time. Every pointer being followed was a legal pointer to a live order object. It was simply the wrong order object, and nothing in the program was in a position to notice.

This is the part I would flag to anyone building on pooled memory. A use-after-free against the system allocator gives you a decent chance of a segfault or a sanitizer report. A use-after-free against your own pool gives you a plausible answer, forever.

3. The hunt

I could not find it by reading the matching logic, because the matching logic was correct. Given the book state it was handed, it did the right thing every time. The state was already wrong when it arrived.

What located it was logging identity rather than values. Every order object got a monotonically increasing generation counter, incremented each time the pool handed it out, and the traversal logged the pool slot together with its generation. The corrupt levels showed the same slot appearing under two different generations in the same book, which is a statement that cannot be true and immediately named the mechanism.

4. Root cause

Cancellation after a partial fill took a path that release on a full fill did not. On a full fill, the order is removed from its price level as part of matching, then released. On cancel, the code found the order, marked it dead, and released it to the pool, and in that path the unlink from the price level list was missing.

```
// cancel path, as written
Order* o = index_.find(order_id);
o->state = OrderState::Dead;
pool_.release(o);            // level_->queue still points here
```

The order is now on the pool free list and simultaneously reachable from the price level queue. Nothing has gone wrong yet. The memory is untouched and a traversal still sees a sane looking dead order.

The damage happens on the next inbound order, which acquires that exact slot, overwrites it with a new price, side, quantity and timestamp, and links it into whichever price level it belongs to. The object now sits in two intrusive lists at once, holding one set of next and previous pointers between them. The old level walks into the new order, reads a price that has nothing to do with that level, and offers it for matching. That is the crossed book. It is also the priority violation, because the new order inherited a position in a queue it never queued for.

The deeper error is a design one. An intrusive node cannot be a member of two lists that share the same link fields, so releasing an object to the pool while it is still linked is not a bug in the cancel path. It is a violation the pool should have been able to refuse.

5. The fix

Two changes, and the second is the one that matters.

First, the cancel path unlinks before releasing, in that order, always.

```
Order* o = index_.find(order_id);
o->level->unlink(o);         // leave every list first
index_.erase(order_id);
o->state = OrderState::Dead;
pool_.release(o);
```

Second, and more usefully, the invariant is enforced at the pool boundary instead of trusted at each call site. release asserts in debug builds that the object is linked nowhere, and acquire asserts that the slot it is handing out is genuinely detached. A null next and previous is a cheap and complete test for this structure.

```
void Pool::release(Order* o) {
    assert(o->next == nullptr && o->prev == nullptr &&
           "order released while still linked into a price level");
    o->generation++;
    free_list_.push(o);
}
```

The generation counter stayed in permanently. It costs four bytes per order and turns any future instance of this class of bug from an invisible wrong answer into something a log can prove, which is worth considerably more than the four bytes.

6. What it changed

The lesson I take from this one is about where invariants live. My cancel path was wrong, and I could have fixed the cancel path and moved on. But the reason the bug was possible is that the pool accepted an object no correct caller would ever hand it, and every future call site would have had the same opportunity to get it wrong. Fixing the caller fixes an instance. Enforcing at the boundary fixes the class.

- Recycling memory yourself removes the crash that would have told you. If you pool objects, you owe yourself the identity checks that the allocator and sanitizers used to provide.
- Any object in an intrusive container has a hard rule: leave the container before you change ownership. Assert it where ownership changes, not where you remember to.
- Two paths that end in the same operation should share the code that gets it right. My fill path and cancel path both ended in release and only one of them unlinked first.
- Silence is not evidence of correctness. This bug produced no crash for as long as I let it run, and it was corrupting the single guarantee the engine exists to provide.

Worth stating plainly for the domain: a matching engine that crashes is an incident, and a matching engine that silently violates price-time priority is a much worse one. The fills it produced looked completely ordinary. Nothing downstream could have detected that they were wrong.