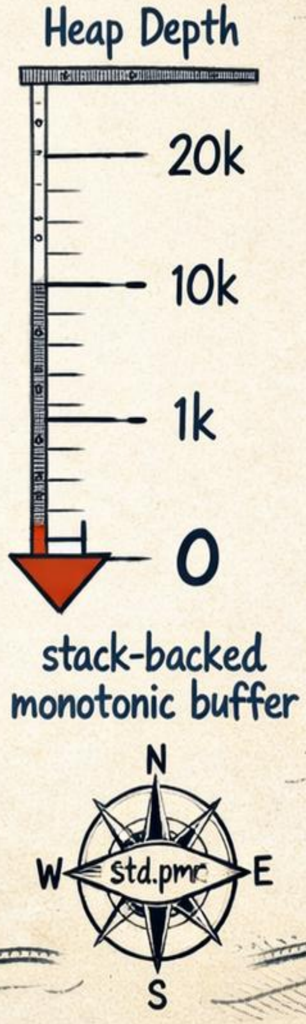
20,000 mallocs → 0 Under the Sea

Practical C++ PMR



C++ User Meeting @GSI/FAIR
Ralph J. Steinhagen, 2026-03-04
R.Steinhagen@GSI.de

Every "new" Has a Cost

A metadata Tag attached to every sample:

- index (std::size_t)
- map (pmr::unoddered_map<string, gr::pmt::Value>)
  - key$_0$ (pmr::string), values$_0$ (scalars, strings, pmr::vector<pmr::string>, ...)
  - key$_1$ (pmr::string), values$_1$
  - ...

Created and destroyed millions of times.
Each construction = multiple heap allocations.

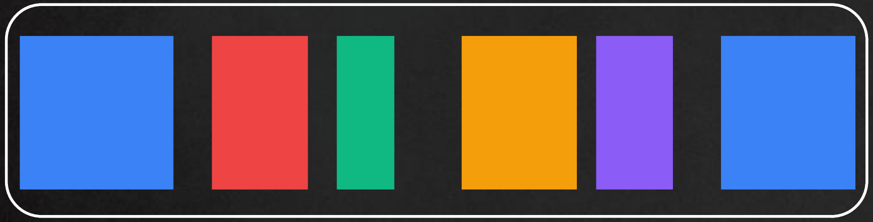
```
using namespace std; // just for slides
using property_map =
          pmr::unordered_map<pmr::string, Value, ...>;


struct Tag {
  std::size_t  index{0UZ};
  property_map map{};
};

// in a tight loop:
for (auto& sample : stream) {
  Tag t{index, {{key, value}, ...}};
  buffer.push(t); // 3+ mallocs per Tag!
}
```

10,000 Tags × 2 allocs = 20,000 calls to malloc
— can we do better?

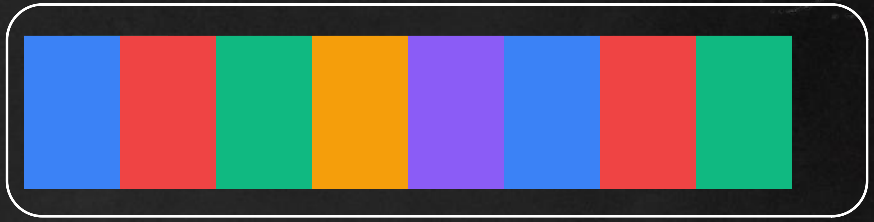
Memory Fragmentation

the silent performance killer

Fragmented Heap (after hours of alloc/free)



Pool / Arena (contiguous, reusable)



✗ Holes between live objects
can't satisfy large requests
✗ TLB pressure, cache misses, page faults
✗ Non-deterministic latency
— poison for real-time

✓ Fixed-size chunks
— no external fragmentation
✓ Contiguous → cache-friendly,
prefetch-friendly
✓ Deterministic: same cost every time

24/7 accelerator control systems run for days to months — the global heap fragments; pools don't.

Why Custom Allocators?

… why we/you should care

1. Cache-Line Alignment
   - SIMD, lock-free buffers, DMA need 64Byte+ alignment.
   - std::allocator gives whatever the heap returns.
2. Allocation Pressure & Latency
   - 20,000 malloc/free per batch dominate wall-clock time.
   - Monotonic bump-pointers eliminate them entirely.
3. Memory Fragmentation
   - Long-running systems fragment the heap.
   - Pool resources recycle fixed-size chunks — no fragmentation.
4. Heterogeneous Computing (Unified / Device Memory)
   - SYCL USM, CUDA unified memory need custom alloc/free.
   - PMR wraps them behind the same interface.

PMR addresses all four.

Cache-Line Alignment

why 64 bytes matters

- Modern CPUs fetch memory in cache lines (typically 64 bytes)
- Misaligned data → false sharing, torn reads, SIMD faults

- **Who needs alignment?**
  - SIMD intrinsics (_mm256_load_ps requires 32B alignment)
  - lock-free ring buffers (head/tail on separate cache lines)
  - DMA transfers (hardware-mandated alignment)
  - GPU shared memory (coalesced access patterns)

- std::allocator guarantees alignof(std::max_align_t)
  → typically 16 bytes. Not enough for any of the above.

Aligned<T>

the old way, done right → e.g. [GNU Radio 4 (GR4)]

```
// gr::allocator::Aligned<T> — from GR4
// 64B alignment → cache-aligned storage
template<typename T, size_t alignment = kCacheLine>
requires(has_single_bit(alignment)
      && alignment >= alignof(T))
struct Aligned {
  using value_type = T;

  T* allocate(size_t n) {
    if (void* p = ::operator new(
          n * sizeof(T), align_val_t{alignment}))
      return static_cast<T*>(p);
    throw std::bad_alloc();
  }

  void deallocate(T* p, size_t) noexcept {
    ::operator delete(p, align_val_t{alignment});
  }
  // + rebind, operator== (see repo)
};
```

Clean, ~20 lines, production-ready.

GR4 uses this for:

- cache-aligned ring buffers
- SIMD-friendly data vectors
- DMA transfer buffers

https://compiler-explorer.com/z/EoPY11Kf3

Aligned<T>

the Problems

```
// template contamination
std::vector<float, gr::Aligned<float>> aligned_data;
std::vector<float>                     normal_data;
// DIFFERENT TYPES!
// → Can't assign or pass them
//   to the same function.

// rebind needed for nested containers:
template<typename U>
struct rebind {
 using other = Aligned<U, alignment>; };

// can't change strategy at runtime:
auto data = make_aligned_vector<float>();
// ← locked to Aligned<float>
// for its entire lifetime
```

1. **Template contamination**

   std::vector<float, Aligned<float>> ≠ std::vector<float>

2. **rebind boilerplate still needed**
3. **Can't swap strategy at runtime**

What if the allocator strategy wasn't part of the type?

Type Erasure to the Rescue

since C++17 std::pmr::

The key insight:

polymorphic_allocator<T> holds a pointer to a `std::memory_resource` (abstract base class with virtual dispatch).

all pmr:: containers share the SAME TYPE regardless of which resource backs them.

old: compile-time policy (template parameter)
new: runtime strategy (virtual dispatch)

```
// Same type, different backends:
pmr::vector<float> a{&stack_resource};
pmr::vector<float> b{&pool_resource};
pmr::vector<float> c{&gpu_resource};

// all three are pmr::vector<float>!
// → assignable, passable,
//   & ABI-stable.

void process(pmr::vector<float>& v);
// ↑ Accepts ANY of the above.
// The resource is the strategy,
// not the type.
```

std::pmr::memory_resource

only 3 virtual methods

```
class memory_resource { // abstract base
protected:
  virtual void* do_allocate(
      size_t bytes, size_t alignment) = 0;

  virtual void do_deallocate(
      void* p, size_t bytes,
      size_t alignment) = 0;

  virtual bool do_is_equal(
      const memory_resource& other)
      const noexcept = 0;
};

// → custom resource in ~30 lines.
// wrap any backing store:
// stack, pool, mmap,
// SYCL USM, CUDA, shared memory, ...
```

Stable ABI:

Works across plugin/library boundaries.
No template instantiation bloat.

Composable:

Resources can chain:
stack → pool → heap → GPU

NVI pattern:

Public allocate() calls protected do_allocate().
Base class handles bookkeeping.

The PMR Zoo

standard resources (C++17)

`std::pmr::new_delete_resource()`

wraps global new/delete. The default.
what you get if you don't specify a resource.

`std::pmr::monotonic_buffer_resource`

bump-pointer allocator. Never frees individually.
Very fast. Perfect for scoped/batch work.

`std::pmr::unsynchronized_pool_resource`

size-bucketed pools. Recycles freed chunks.
No fragmentation. Single-threaded.

`std::pmr::synchronized_pool_resource`

thread-safe variant of the above.
Slightly more overhead (internal locking).

`std::pmr::null_memory_resource()`

Always throws `std::bad_alloc`.
Budget enforcer & overflow guard.

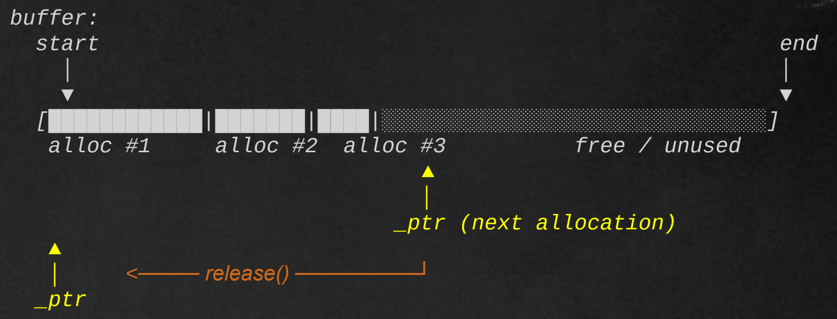
Also: get/set_default_resource() — process-wide default for all pmr:: containers

std::pmr::monotonic_buffer_resource

the bump pointer

How it works:

1. You give it a buffer (stack or heap)
2. Each allocate() bumps a pointer forward
3. Individual deallocate() is a no-op
4. release() frees everything at once



```
buffer:
  start                                                     end
    |                                                        |
    ▼                                                        ▼
 [                                                           ]
  alloc #1      alloc #2  alloc #3           free / unused
                               ▲
                               |
                           _ptr (next allocation)
   ▲
   |         <——— release() ————————┘
 _ptr
```

```
allocate(n):
  - returns _ptr
  - advances _ptr by n (plus alignment padding)

deallocate(p, n):
  - no-op (memory is not reclaimed individually)

release():
  - resets _ptr back to start (all allocations reusable at once)
```

Cost per allocation:

increment a pointer.

That's it. No free lists, no locking, no metadata.

Perfect when:

- All objects have the same lifetime
- You can estimate max memory needed
- Latency matters more than memory efficiency

std::pmr::monotonic_buffer_resource

Zero Heap Allocs: stack buffer + monotonic

```cpp
// 1 KByte on the stack — plenty for a handful of short strings
std::array<std::byte, 1024UZ> buf{};
std::pmr::monotonic_buffer_resource mbr{ buf.data(), buf.size(),
    std::pmr::null_memory_resource()   // overflow → exception!
};

// all allocations come from 'buf' — zero malloc calls
Tag tag{42UZ, {{"BEAM:POSITION:X", 1e4f},
                        { "ctx", {"FAIR.ACCELERATOR", "INJECTION", "CYCLE.2024.Q3"}}},
        &mbr};

// when 'mbr' goes out of scope → all memory reclaimed at once.
// no individual deallocations. That's the monotonic contract.
```

Monotonic = bump pointer · No per-object free ·
null_resource as guard · Perfect for short-lived scoped work

Lifetime Phases

Build → Use → Discard

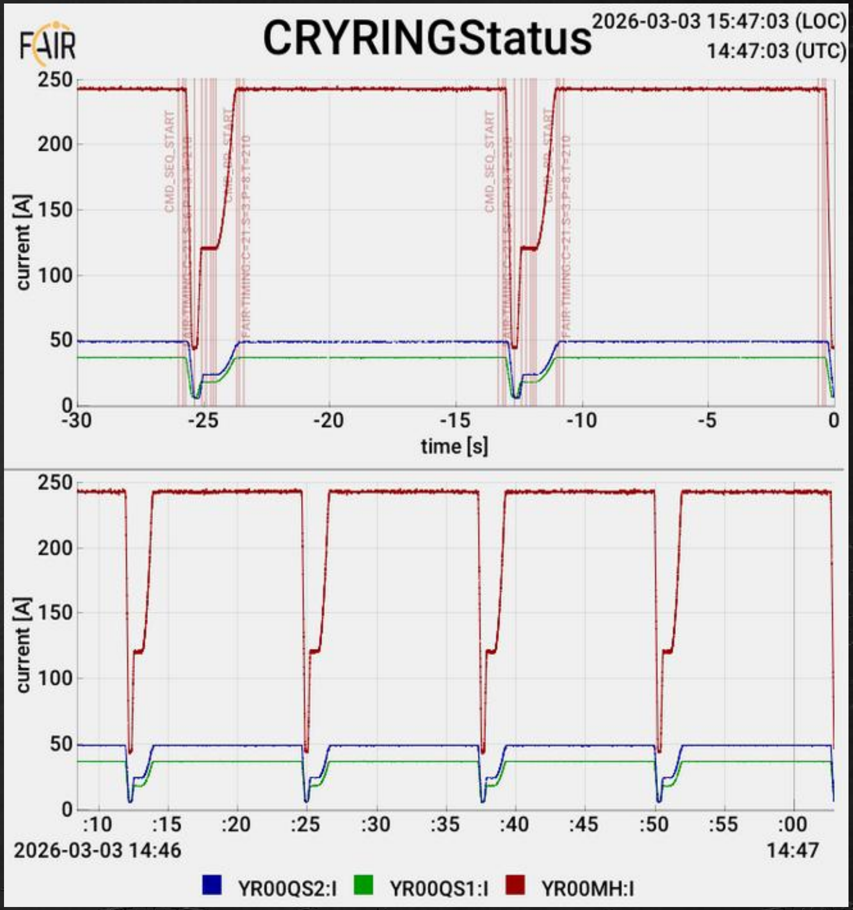
Many workloads have clear phases:

1. **Build** — allocate temp structures
2. **Use** — process the data
3. **Discard** — free everything at once

In accelerator control:

Each acquisition cycle builds Tags, attaches them to samples, then the cycle ends and all metadata is discarded.

→ One arena per cycle, release() at end. No individual deletes. No fragmentation.



https://yr00mh.cryring.gsi.de/web/index.html#dashboard=CRYRINGStatus

Per-Cycle Arena

example

```
// Per-acquisition-cycle arena
pmr::monotonic_buffer_resource cycle_arena{ 8192UZ, new_delete_resource()};

// ── build phase ──
pmr::vector<Tag> tags{&cycle_arena};
tags.emplace_back("BEAM:POS:X", "mm");
tags.emplace_back("RF:FREQ", "Hz");
// all allocs from cycle_arena — fast!

// ── Use phase ──
for (auto& tag : tags)
    process(tag);

// ── Discard phase ──
tags.clear();
cycle_arena.release(); // FREE EVERYTHING, zero fragmentation, ready for next cycle
```

In GR4: target one arena per acquisition cycle.
10K Tags, zero fragmentation, microsecond teardown.

https://compiler-explorer.com/z/qh8nE8rqv

Fallback Pattern I

guard mode (fail-fast)

```
// Development: strict — catch overflow early
std::array<std::byte, 512UZ> buf{};
std::pmr::monotonic_buffer_resource strict{ buf.data(), buf.size(),
    std::pmr::null_memory_resource()   // BOOM if overflow
};

std::pmr::vector<int> v{&strict};
v.assign({1, 2, 3, 4, 5});  // OK — fits in 512 bytes

// v.resize(10000);  // → throws std::bad_alloc!
// null_memory_resource says: "you sized your buffer wrong"
```

`std::pmr::null_memory_resource()` as the upstream resource.

If the buffer overflows, you get an immediate exception.
This catches sizing bugs during development and testing.

Think of `std::pmr::null_memory_resource` as a memory budget enforcer

https://compiler-explorer.com/z/sWMhjqGeM

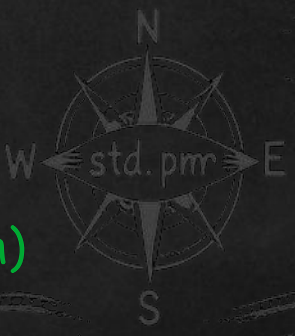
Fallback Pattern II

graceful overflow

```
// production: graceful — heap as safety net
std::array<std::byte, 512UZ> buf{};
std::pmr::monotonic_buffer_resource safe{ buf.data(), buf.size(),
    std::pmr::new_delete_resource()    // overflow → malloc
};

std::pmr::vector<int> v{&safe};
v.resize(10000);  // overflows stack buffer → falls back to heap
// fast path stays on stack. Only unusual payloads hit malloc.
```

Best of both worlds:

- Normal case: stack buffer handles everything → zero malloc
- Unusual case: graceful overflow to heap → no crash



Development: use std::pmr::null_resource (catch bugs)

Production: use std::pmr::new_delete_resource (never crash)

https://compiler-explorer.com/z/sWMhjqGeM

Measuring Fallback Rate

how often do you hit the slow path?

```
// Use CountingResource as upstream to measure overflow
CountingResource overflow_counter;
std::array<std::byte, 64UZ> tiny{};
std::pmr::monotonic_buffer_resource measured{ tiny.data(), tiny.size(),
    &overflow_counter};

std::pmr::vector<int> v{&measured};
v.resize(1000);

// overflow_counter.allocations tells you
// how many times you hit the heap.
// → If non-zero in typical usage: grow your buffer.
```

If your fallback counter is non-zero in normal workloads, your buffer is too small. Measure, then size.

CountingResource

allocation tracker (~30 lines)

```cpp
struct CountingResource : std::pmr::memory_resource {
  std::pmr::memory_resource* upstream{new_delete_resource()};
  size_t allocations{0}, deallocations{0};
  size_t bytes_allocated{0}, bytes_deallocated{0};

  void* do_allocate(size_t b, size_t a) override {
    ++allocations; bytes_allocated += b;
    return upstream->allocate(b, a);
  }
  void do_deallocate(void* p, size_t b, size_t a) override {
    ++deallocations; bytes_deallocated += b;
    upstream->deallocate(p, b, a);
  }
  bool do_is_equal(const memory_resource& o)
    const noexcept override { return this == &o; }

  bool is_balanced() const {                       // ← key method
    return allocations == deallocations
           && bytes_allocated == bytes_deallocated;
  }
};
```

https://compiler-explorer.com/z/snoqbsYTo

CountingResource
usage & output

```
CountingResource mr;
{
    std::pmr::vector<int> v{&mr};
    v.assign({1, 2, 3, 4, 5});
    mr.print("in-scope");
}
mr.print("exited");
assert(mr.is_balanced());
```

Use in unit tests:

- verify zero leaks for every code path
- track allocation counts (regression testing)
- measure bytes consumed per operation

Output:

```
[in-scope] a=2 d=1 bytes=20/0 PENDING
[exited  ] a=2 d=2 bytes=40/40 BALANCED
assert passed — no leaks
```

https://compiler-explorer.com/z/snoqbsYTo

When Objects Come and Go

std::pmr::pool_resource

`std::pmr::monotonic`

- everything dies together (scoped/batch).
- no individual free. Blazing fast.

`std::pmr::[un]synchronized_pool_resource`

- recycles freed chunks in size-buckets.
- no heap fragmentation over time.
- (optional: thread-safe variant. slightly more overhead)

Key for long-running systems:

- Accelerator control loops run 24/7 for months.
- The global heap fragments; pools don't.

Stacked Resources

pool over monotonic

```
// two-level stacking: powerful pattern!
std::array<std::byte, 8192UZ> buf{};
std::pmr::monotonic_buffer_resource mono{ buf.data(), buf.size()};
std::pmr::unsynchronized_pool_resource pool{&mono};
//                                     ^^^^
// pool handles reuse;
// monotonic handles pool's bookkeeping.

std::pmr::map<std::pmr::string, double> signals{&pool};
signals["BEAM:POS:X"] = 3.14;
signals.erase("BEAM:POS:X");
signals["BEAM:POS:X"] = 2.71;
// ↑ reuses freed memory — no heap call!
```

Pool recycles the chunks; monotonic provides the backing store.

When the monotonic buffer goes out of scope, everything is freed.

Benchmarks & Measurements

~ Surfacing for Air ~

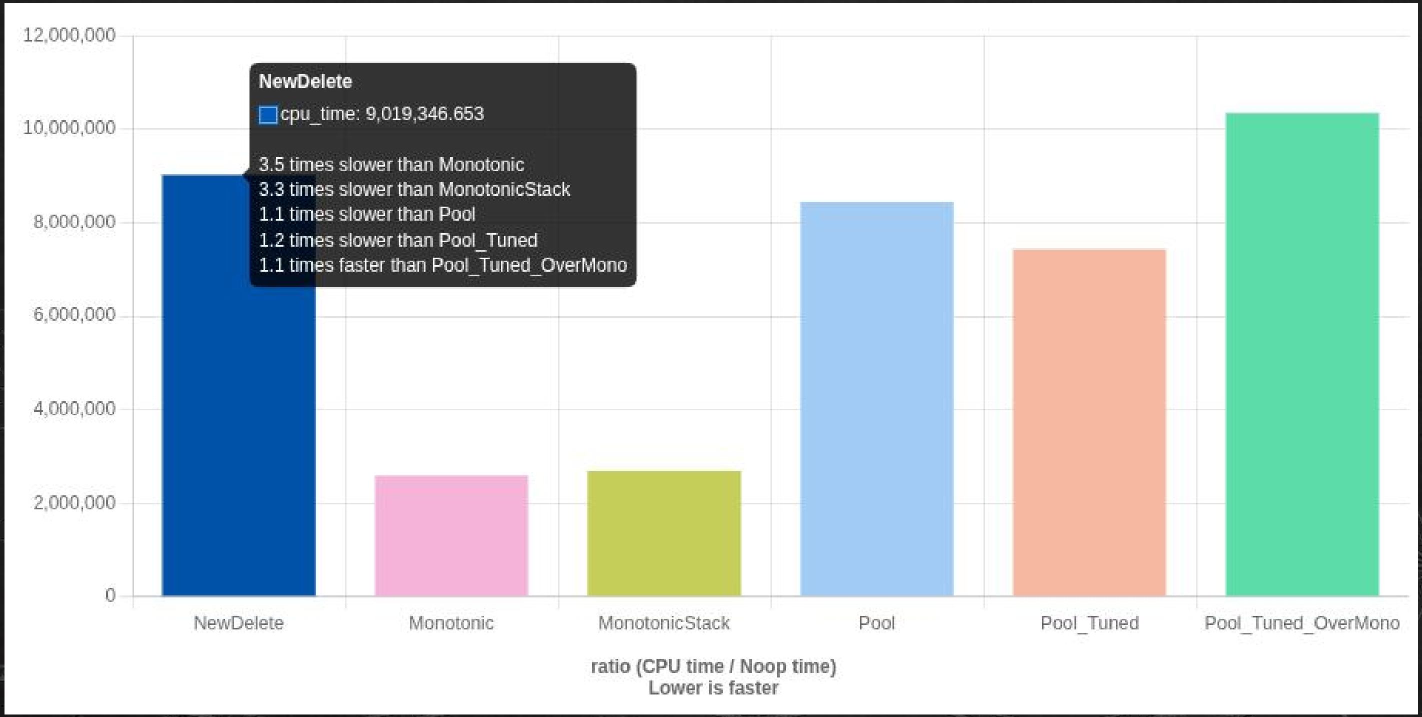
10’000 Tag Constructions

https://compiler-explorer.com/z/7x53W94b7

20’000 heap allocs (baseline)

0 heap allocs (monotonic)

5 heap allocs (pool)



NewDelete

■ cpu_time: 9,019,346.653

3.5 times slower than Monotonic
3.3 times slower than MonotonicStack
1.1 times slower than Pool
1.2 times slower than Pool_Tuned
1.1 times faster than Pool_Tuned_OverMono

https://quick-bench.com/q/5JRxaSw4j8Hk-4f-iGaFiVUTM4Y

gr::pmr::migrate<T>

bridging memory resources

Problem:

Data lives on resource A, but you need it on resource B.

(e.g., host memory → GPU shared memory)

Solution:

Allocate on target, move/copy data, free source.

exception-safe. Compile-time dispatch for speed.

Three paths (compile-time):

`std::trivially_copyable` → `std::memcpy` (fastest)

float, double, int, ...

`std::nothrow_move` → `std::uninitialized_move_n`

std::string, Tag, ...

throwing copy → copy + rollback (strong guarantee)

gr::pmr::migrate<T>

implementation (from GR4)

```
template<class T>
T* migrate(memory_resource& target, memory_resource& source,
           T* ptr, size_t count) {
  if (&target == &source) return ptr;            // no-op
  T* dest = static_cast<T*>(
    target.allocate(count*sizeof(T), alignof(T)));
  try {
    if constexpr (is_trivially_copyable_v<T>)
      memcpy(dest, ptr, count * sizeof(T));      // fast
    else if constexpr (is_nothrow_move_constructible_v<T>)
      uninitialized_move_n(ptr, count, dest);    // move
    else
      for (size_t i=0; i<count; ++i)
        construct_at(dest + i, ptr[i]);          // safe
  } catch (...) {
    destroy_n(dest, constructed);
    target.deallocate(dest, ...); throw;         // rollback
  }
  destroy_n(ptr, count);
  source.deallocate(ptr, ...);
  return dest;                                   // done
}
```

https://compiler-explorer.com/z/9Pb99Yqxv

gr::pmr::migrate<T>

implementation (from GR4)

```cpp
CountingResource host_mr, device_mr;
constexpr size_t N = 1000;

// allocate on host
auto* host_ptr = static_cast<double*>(
    host_mr.allocate(N * sizeof(double)));
for (size_t i = 0; i < N; ++i) host_ptr[i] = double(i);

// Migrate to "device" (in production: SYCL USM resource)
auto* device_ptr = gr::allocator::pmr::migrate<double>(
    device_mr, host_mr, host_ptr, N);

// host_mr is balanced (source freed), device_mr owns data
assert(host_mr.is_balanced());
assert(device_ptr[42] == 42.0);
```

In production: host_mr & device_mr are backed by SYCL/CUDA resources.

https://compiler-explorer.com/z/9Pb99Yqxv

STL Allocator vs. PMR

… how to choose.

STL Allocator (e.g. Aligned<T>)

Compile-time policy

- ✓ zero virtual dispatch overhead
- ✓ guarantees alignment at type level
- ✓ inlineable — visible to optimiser
- ✗ infects every container type
- ✗ can't change strategy at runtime
- ✗ rebind boilerplate

GR4: cache-aligned buffers/SIMD

PMR (polymorphic_allocator)

Runtime strategy

- ✓ one type for all backends
- ✓ swap host/device at runtime
- ✓ composable (stack resources)
- ✗ virtual dispatch per alloc (~1-3ns)
- ✗ alignment not encoded in type

GR4 use: CircularBuffer, Tag metadata

PMR Wins When...

- Many small allocations in hot paths
  - (Tags, messages, temp buffers)
- Clear lifetime phases: build → use → discard
  - (acquisition cycles, request handling, batch processing)
- API stability matters
  - (plugin boundaries, shared libraries, ABI compatibility)
- Memory accounting is needed
  - (CountingResource, budget enforcement, leak detection)
- Multiple memory backends
  - (host, GPU, NUMA, shared memory — same container type)
- Long-running systems where fragmentation matters
  - (accelerator control, embedded systems, servers)

Device Memory & Beyond

~ Diving Deeper ~

Unified Memory — Host ↔ Device

SYCL USM as a memory_resource

```
struct usm_shared_resource : public std::pmr::memory_resource {
  sycl::queue& _q;

  void* do_allocate(size_t bytes, size_t align) override {
    return sycl::aligned_alloc_shared( align, bytes, _q);
  }

  void do_deallocate(void* p, size_t, size_t) override { sycl::free(p, _q);}
  // do_is_equal: compare queue contexts
};
```

Wrap vendor API as memory_resource:

SYCL USM: `sycl::malloc_shared / free`
CUDA: `cudaMallocManaged / cudaFree`
OpenCL: `clSVMAlloc / clSVMFree`

```
// migrate host → device in one call:
auto* dev = gr::pmr::migrate<float>(
    usm_resource, host_resource,
host_ptr, N);

// or using USM directly
```

std::pmr::vector<float> doesn't know whether it lives on CPU (host) or GPU (device).

NUMA & Huge Pages

same interface, different backing

```
// huge-page resource (Linux)
struct huge_page_resource
    : public std::pmr::memory_resource {

  void* do_allocate(size_t bytes, size_t) override {
    // 2 MiB
    size_t sz = round_up(bytes, 1UL << 21);
    void* p = mmap(nullptr, sz,
        PROT_READ | PROT_WRITE,
        MAP_PRIVATE | MAP_ANONYMOUS
        | MAP_HUGETLB, -1, 0);

    if (p == MAP_FAILED) throw bad_alloc();
    return p;
  }
  void do_deallocate(void* p, size_t bytes, size_t)
    override {
    munmap(p, round_up(bytes, 1UL << 21));
  }
};

// Usage: same containers, different backing
std::pmr::vector<float> buf{&huge_resource};
```

The pattern extends to:

huge pages (2 MiB / 1 GiB)

fewer TLB misses for large buffers.

NUMA-local allocation

pin to socket running the thread.
mbind() behind do_allocate.

Shared memory (IPC)

mmap a shared fd. Multiple processes
use the same pmr:: containers.

GR4: Double-mmap-ed
CircularBuffer<T>

containers don't change -- only backing resources

What to Remember

1 **Alignment → Aligned<T> or aligned PMR resource**

Cache-line alignment for SIMD, lock-free buffers, DMA.

2 **Latency → monotonic_buffer_resource**

Bump-pointer: 20,000 → 0 heap allocs. Stack buffer + null_resource as guard.

3 **Fragmentation → pool_resource**

Long-running systems. Pool recycles fixed-size chunks — deterministic, zero fragmentation.

4 **Device memory → custom memory_resource**

Wrap SYCL USM / CUDA unified. pmr::migrate<T>() moves data host↔device.

measure – measure – measure – avoid PMO (premature optimisation)!

Summary

What to Remember

1 **PMR = practical custom allocators**

Type erasure via memory_resource. All pmr:: containers share one type.
No more template contamination.

2 **monotonic_buffer_resource first**

Your go-to for anything short-lived. Stack buffer + `std::pmr::null_resource` as guard.
(20,000 → 0 heap allocs)

3 **Build your own for exotic backends**

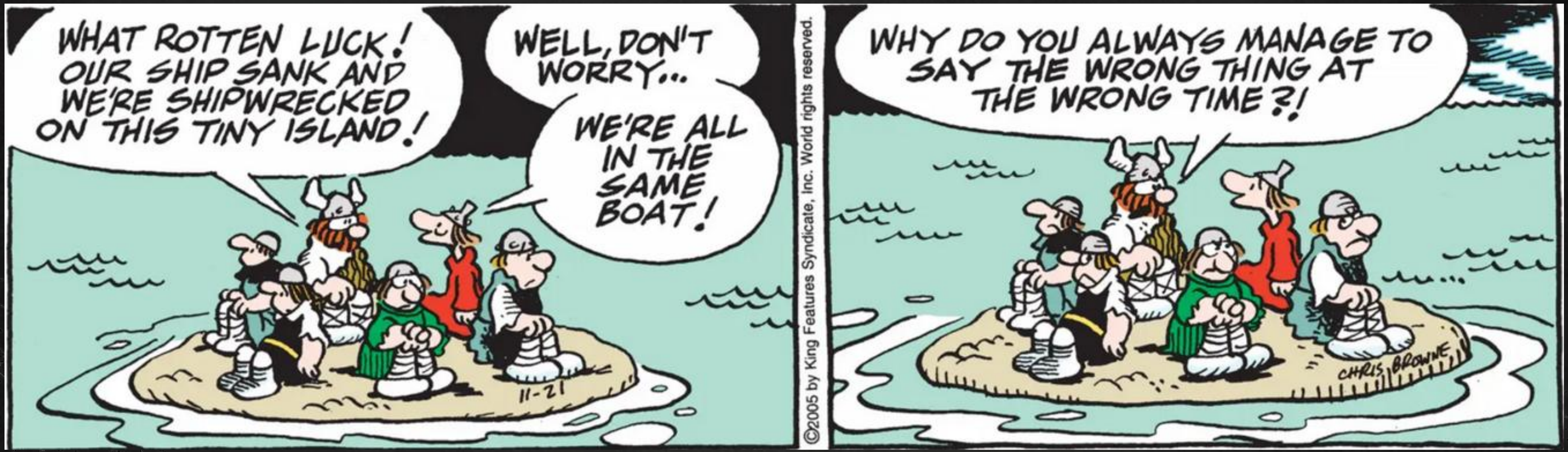
Only 3 virtual methods. Wrap SYCL USM, CUDA managed memory, `mmap` — the sky is the limit.

4 **Test with CountingResource + is_balanced()**

Verify zero leaks. Use pmr::migrate<T>() to move data between resources with strong exception guarantee.

measure – measure – measure – avoid PMO (premature optimisation)!

That's all – Questions?


WHAT ROTTEN LUCK! OUR SHIP SANK AND WE'RE SHIPWRECKED ON THIS TINY ISLAND!
WELL, DON'T WORRY..
WE'RE ALL IN THE SAME BOAT!
11-21
WHY DO YOU ALWAYS MANAGE TO SAY THE WRONG THING AT THE WRONG TIME?!
CHRIS BROWNE

©2005 by King Features Syndicate, Inc. World rights reserved.