Two advanced PMR techniques in C++17/20

MAREK KRAJEWSKI

whoami()

- C++ (et al.)
- Networking protocols
- Client-Server
- Qt and UI
- Linux, Windows, Embedded
- Freelancer
- Concerned about performance

Hands-On
High Performance
Programming
with Qt 5

Packt

Overview

- Intro: memory allocators for performance and more
- STL Allocators vs PMRs
- Advanced techniques*
  - Wink-Out (and/or arenas)
  - Localized GC (aka. self-contained heaps)
- Some lessons

* *C++ Weekly, Ep.250: Custom Allocation - How, Why, Where (Huge multi threaded gains and more!), https://www.youtube.com/watch?v=5VrX_EXYIaM*

Memory and performance

- Allocating Memory – costly!
- Accessing Memory – slow!
  - cache hierarchy
  - cache invalidation (& importance of locality)

→ Let‘s keep that talk on the intermediate level!

- System Memory Allocators (*ptmalloc*, *tcmalloc*, *jemalloc, NT Heap, mimalloc, Mesh**)
- Custom Memory Allocators

* *ptmalloc – GNU, tcmalloc – Google, jemalloc – Facebook, mimalloc- Microsoft, etc.*

Why custom memory allocators?

1. For performance!
   i. Allocate related objects in contiguous memory
      - for locality - fighting fragmentation / diffusion → *Mesh*, a heap compacting allocator
      - for separation (e.g. hot/cold data separation)

   ii. Avoid locks when in the same thread*, lower costs of allocations/deallocations in general

2. For special placement
   - i.e. in special memory region: shared memory, file-mapped memory, high-BW memory, persistent memory, even program stack!

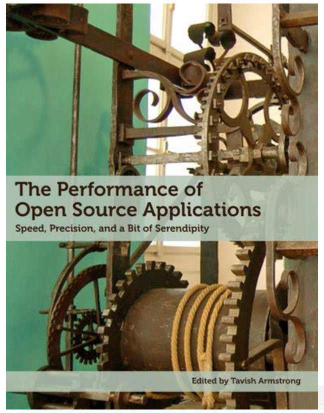
3. For security
   - using tagged/protected memory or special heaps (e.g. *PartitionAlloc* in Chrome to combat UaF** bugs)

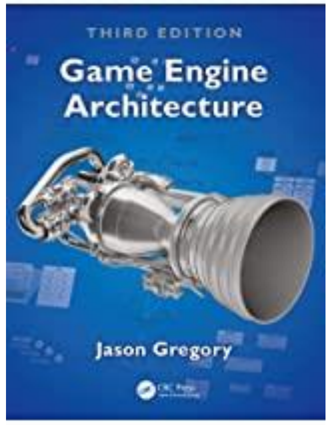
4. For debugging
   - e.g. print some debug info when allocating

___
* however: *tcmalloc* / *jemalloc* / *gcc malloc* already use thread-local memory buffers!!!
* **UaF – Use after Free*

Traditional allocator wisdom

- 1st usage I'ʻve seen - sized blocks allocators
  - e.g. for network packets: overloading C++ *new() operator* in a class
  - arena – a buffer where objects will be placed
- XML parsing using lists for DOM representation [*Book 1*]
  - *Buffer-backed bump-up allocator* (but it can release elements)
  - List'ʻs performance problems solved!
- Game programming community: the classic [*Book 2*]
  - *stack(ed) allocators* – memory cannot be freed in arbitrary order! (bump-up/down)
  - *pool allocators* - particles, projectiles, spaceships, etc.
  - *aligned allocators* – e.g. for optimal DMA throughput
  - *single-frame allocators* – at the begin of each frame stacks top poiner is resetted!


The Performance of Open Source Applications
Speed, Precision, and a Bit of Serendipity
Edited by Tavish Armstrong



„Jak & Daxter“ game analysis

The loader generally tracks 3 different memory heaps -- the common heap, and two level heaps. The game ***has two levels loaded at any one time****, each getting their* ***own self-contained heap*** *(this is what allows you to seamlessly walk from one to another).*

The common heap *contains the core engine code and data.*

The loader ***just uses a simple bump allocator*** *to throw new data on the end. Once the level is finished with, it just throws out the entire heap and starts again*

STEPHEN WHITE, NAUGHTY DOG, INC.

Allocators vs PMRs

STL Allocators

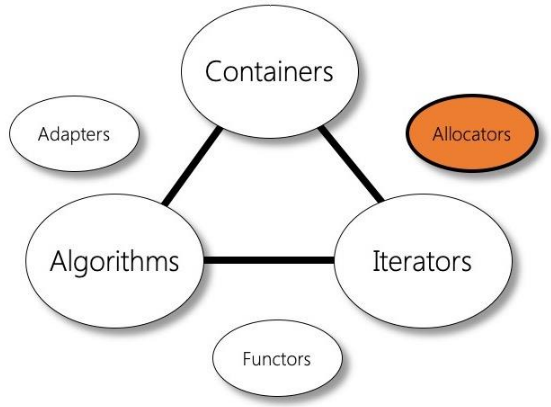
- C++ 98/03 & STL – as template parameter

```
template<class T, class Allocator = allocator<T>> class vector;
```

- Where did they come from?

  - *first*: to encapsulate different size pointers on PCs at the time (*near/far/huge* pointers, segmented memory model, …)

  - *later*: abstract memory allocation for containers



- Because:

  - containers need an interface that is more “granular” than *new* and *delete*!
  - i.e. *allocate/deallocate*, *construct/destroy*

C++89/03 Allocator problems

- The original design was technically lacking in several aspects
  - Because of time pressure (?!)
- Allocator is part of container‘s type

```
void func(const std::vector<int>& v);

std::vector<int> vec();
std::vector<int, MyAllocClass<int>> myvec(myAlloc);

func(vec);   // OK
func(myvec); // compiler ERROR !!!

// must/could be:
template <class Alloc> void func(const std::vector<int, Alloc>& v); // WTF ???
```

- PITA !!! - excuse me, but that‘s not scalable…
  - However: this could be a good thing if pointers were of different size/type

Bloomberg design to the rescue

- Before STL/C++98:
  - They had developed a set of stateful, generally non-equal allocators before STL*
  - Not part of container‘s type! - passed by pointer to each allocator-aware (AA) class
  - They tried to merge both worlds, finally they joined the committee to work on allocators (*2005*)
- The idea: (P2126R0, N1850=05-0110)
  - wrap the base class for allocators in an STL-conformant *Allocator* wrapper
  - then always use the wrapper in the signatures of the STL containers
  - but also add allocator as parameter in constructors of all classes
    - ➢ *bsl::string, bsl::vector, bsl::list…*- i.e. need for extended *std::* classes
    - ➢ thus it can be passed down the chain in e.g. *bsl::vector< bsl::list< bsl::string* > > > !!!

___
* known internally as „*Lakos allocator model*“

C++11 Allocators

- We got some changes in C++11 fixing the technical issues:

  → added scoped and stateful allocator support

  → simplified basic allocator requirements, provided a template impl. for most requirements!

  → but also: added more traits & mechanisms
- However, allocators were still part of type signature of containers (again, no time…)
- Remember?
  a) WTF?
  b) PITA!
  c) Etc…
- In C++11 some newly designed classes used *„type erased“* allocators
  - *std::function , std::promise, std::shared_ptr*
  - In C++17 containers didn‘t follow that track

Bloomberg design in C++17

namespace **std** {

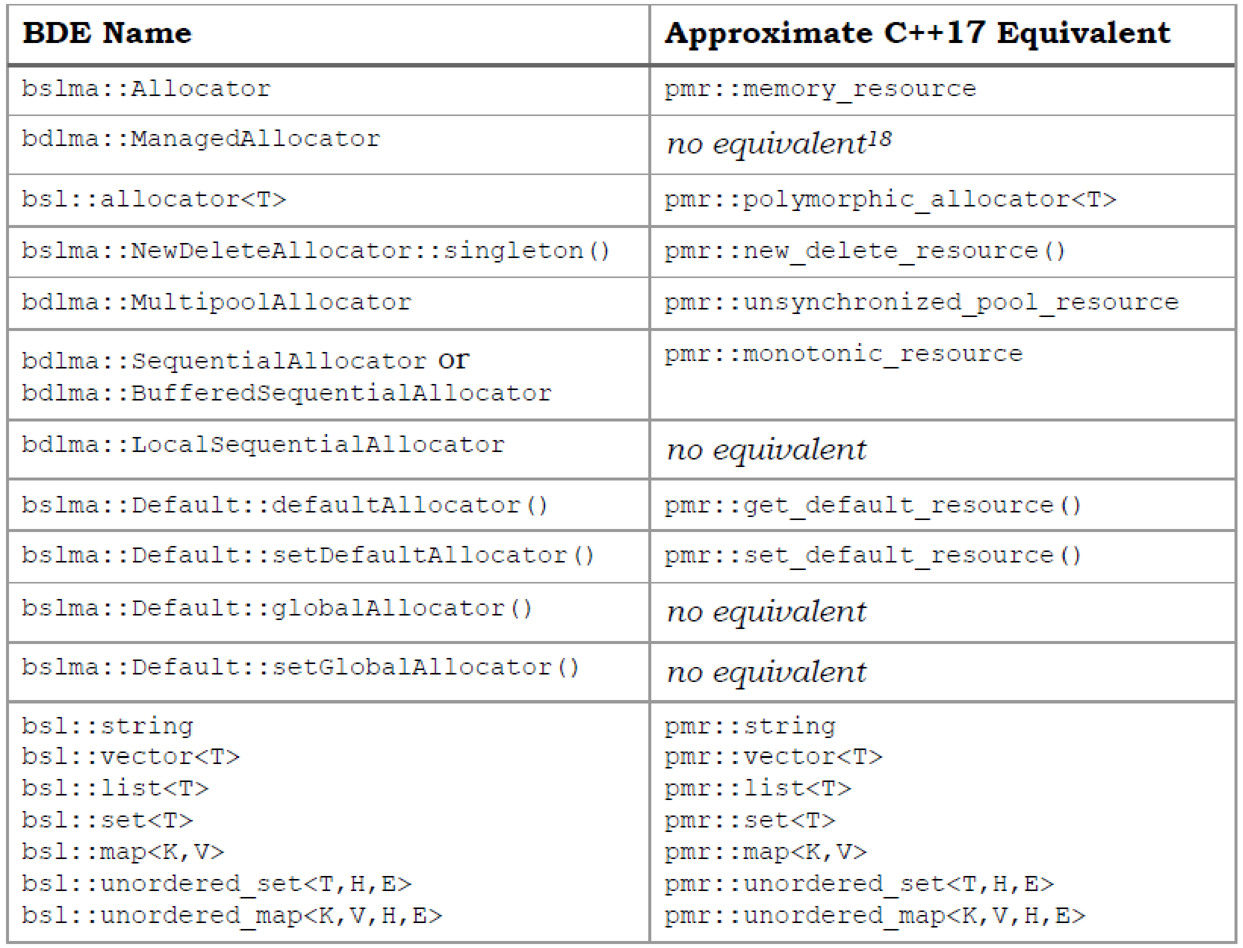
…

namespace **pmr** {

template <class T> using **vector** =
  std::vector<
    T,
    polymorphic_allocator<T>
  >;

}}

Table 2: BDE to C++17 name mappings



BDE Name	Approximate C++17 Equivalent
`bslma::Allocator`	`pmr::memory_resource`
`bdlma::ManagedAllocator`	*no equivalent*[18]
`bsl::allocator<T>`	`pmr::polymorphic_allocator<T>`
`bslma::NewDeleteAllocator::singleton()`	`pmr::new_delete_resource()`
`bdlma::MultipoolAllocator`	`pmr::unsynchronized_pool_resource`
`bdlma::SequentialAllocator` or `bdlma::BufferedSequentialAllocator`	`pmr::monotonic_resource`
`bdlma::LocalSequentialAllocator`	*no equivalent*
`bslma::Default::defaultAllocator()`	`pmr::get_default_resource()`
`bslma::Default::setDefaultAllocator()`	`pmr::set_default_resource()`
`bslma::Default::globalAllocator()`	*no equivalent*
`bslma::Default::setGlobalAllocator()`	*no equivalent*

`bsl::string`
`bsl::vector<T>`
`bsl::list<T>`
`bsl::set<T>`
`bsl::map<K,V>`
`bsl::unordered_set<T,H,E>`
`bsl::unordered_map<K,V,H,E>`
	`pmr::string`
	`pmr::vector<T>`
	`pmr::list<T>`
	`pmr::set<T>`
	`pmr::map<K,V>`
	`pmr::unordered_set<T,H,E>`
	`pmr::unordered_map<K,V,H,E>`

Enter PMRs!

What is a PMR?

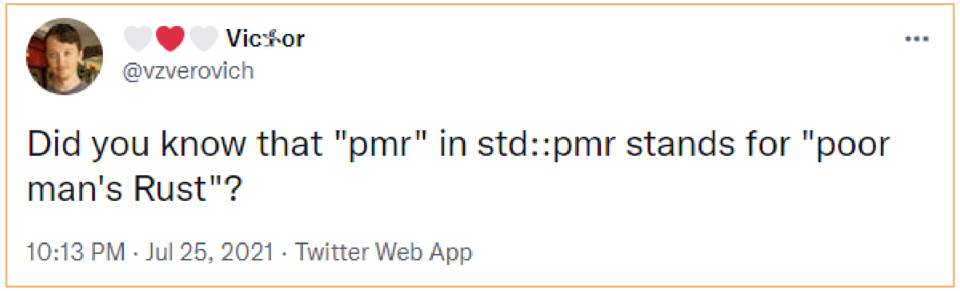
→ poor man‘s Rust? ☺

aka

→ Polymorphic Memory Resource

Wait …

→ why not Polymorphic Allocator?

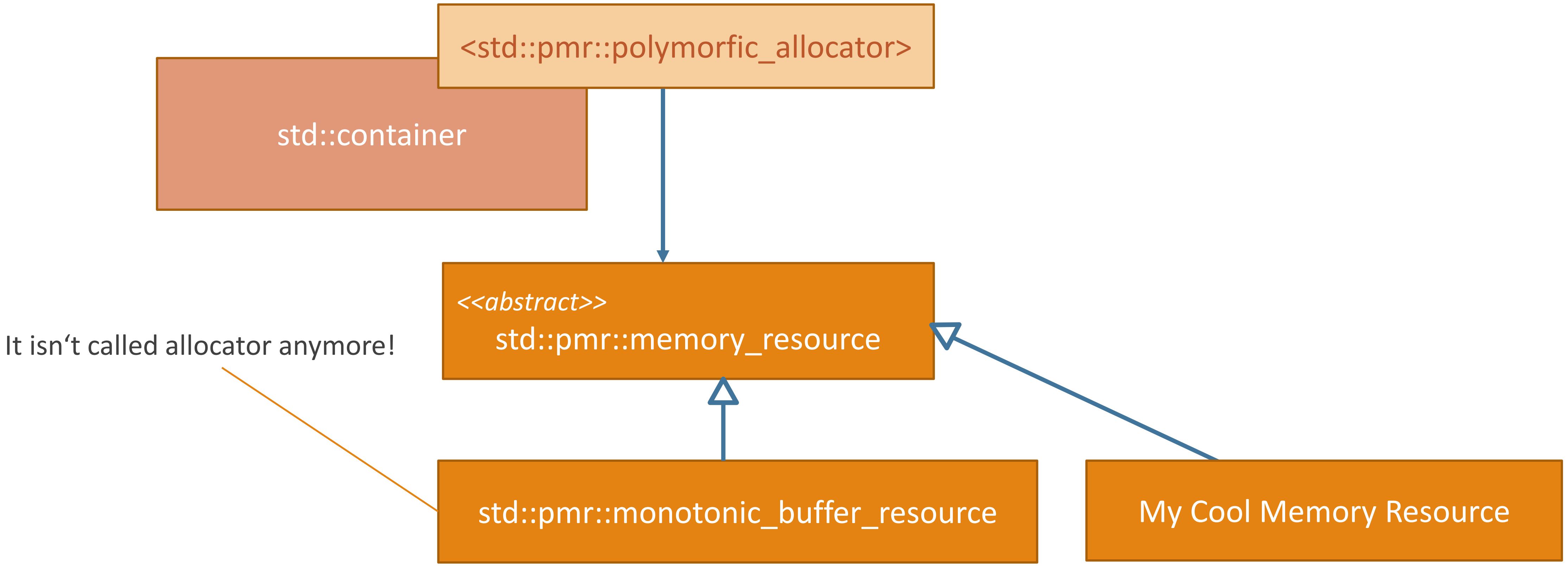


Victor
@vzverovich

Did you know that "pmr" in std::pmr stands for "poor man's Rust"?

10:13 PM · Jul 25, 2021 · Twitter Web App

C++17 Polymorfic Memory Resources



std::container
<std::pmr::polymorfic_allocator>
<<abstract>>
std::pmr::memory_resource
It isn‘t called allocator anymore!
std::pmr::monotonic_buffer_resource
My Cool Memory Resource

A Closer Look at PMRs

Available PMR types

- *std::pmr::monotonic_buffer_resource*
  - releases the allocated memory only when the resource is destroyed
  - stacked allocator (almost!), on-stack allocator, single-frame allocator
  - In maths lingo: monotonic = always growing, i.e. elements rarely deleted!
  - Good for data structs that are buitlt up at once
- *std::unsynchronized_pool_resource*
  - optimized for blocks of single size
  - pool allocator - it consists of a collection of *pools* that serves requests for different block sizes.
  - Good for data struct with numerous insertions/deletions (memory reuse, good locality)
- *std::synchronized_pool_resource*
  - i.e. pool allocator with locks
- *std::pmr::pool_options*
  - can parametrize the pool resource

Monotonic resource example

- Basic usage

```
unsigned buff[1024] = {};
std::pmr::monotonic_buffer_resource buffer_mem_res(buff, sizeof(buff));

std::pmr::vector<std::pmr::string> vec1(&buffer_mem_res); // forwards the allocator
std::pmr::vector<int> vec2(&buffer_mem_res);
```

- Caveat: *std::pmr::monotonic_buffer_resource* in loops

```
for (int i = 0; i < N; ++i)
{
  buffer_mem_res.release(); // rewind to the beginning!

  std::pmr::vector<std::pmr::string> strg_vec(&buffer_mem_res);
  strg_vec.push_back("strg XXX");

  // etc...
}
```

Pool resource example implementation

- Figure taken from N3916 r2 →
- Configuration points visible
- Each pool manages a collection of *chunks* that are then divided into blocks of uniform size.
- Good locality

[*Example:* Figure 1 shows a possible data structure that implements a pool resource.

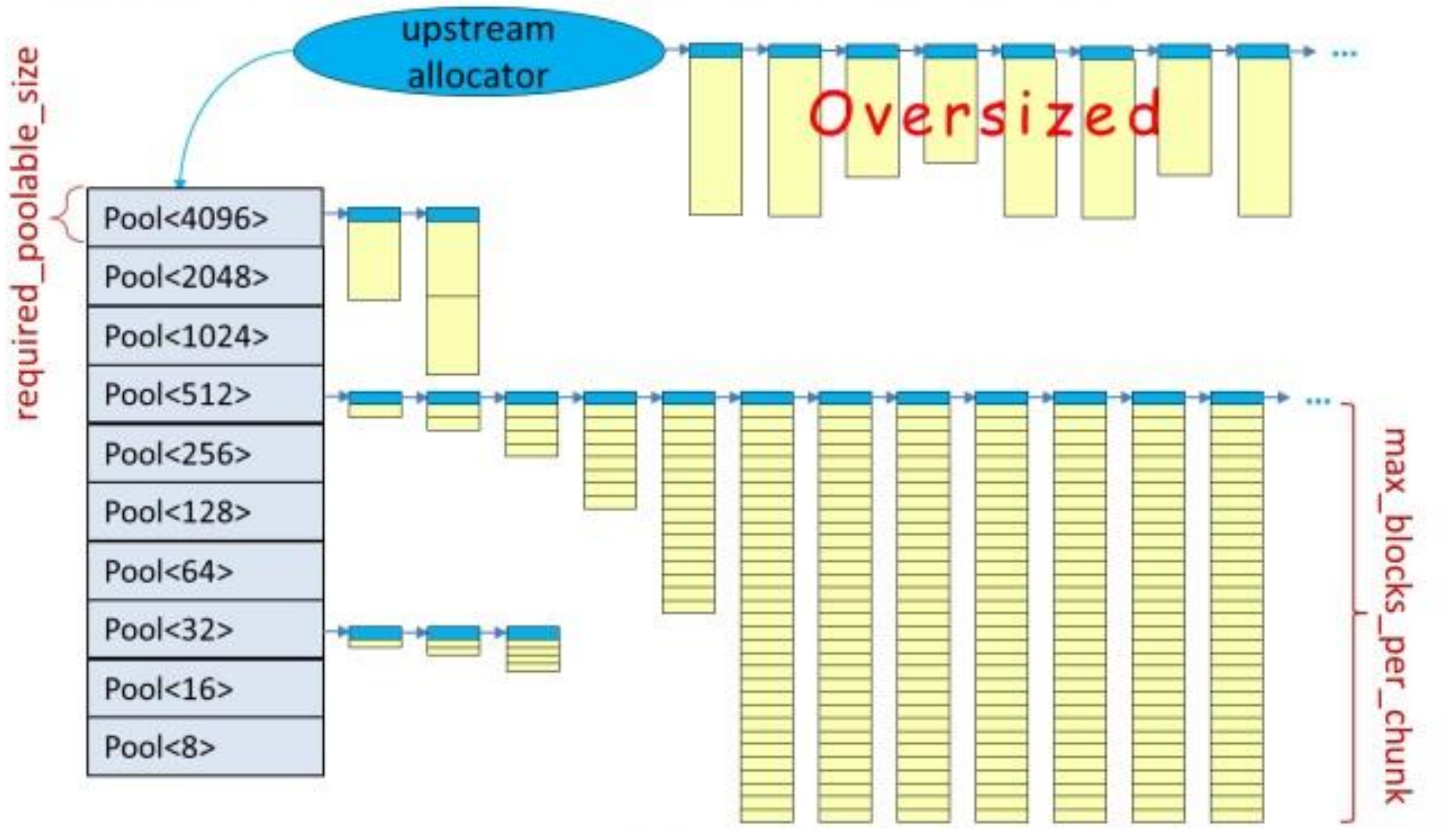


Figure 1: pool resource

– end example]

ib-krajewski.de

Upstream allocators/resources

- We can set a fallback (aka upstream) memory resource
  - the current resource will get memory when it runs out of ist own!
- Where defined?
  - in each *std::* memory resource‘s constructor (but not in the base class!), e.g.:

```
explicit monotonic_buffer_resource(std::pmr::memory_resource* upstream);
```

  - For example set a *monotonic_buffer_resource* as an upstream for *unsynchronized_pool_resource*
- Default upstream:
  - *std::pmr::new_delete_resource()*
- Bloomberg - setting the default and global allocators
  - In C++17: only the default resource
  - *std::pmr::set_default_resource(std::pmr::memory_resource* r)*
  - Why change it? - for testing and debugging

Wink-Out

Wink-Out

- The *deallocate()* method in *pmr::monotonic_buffer_resource* is a no-op!
  - i.e. no destructor for the elements called – time savings!
- When used in container: still container‘s destr will be called… ☹
- Trick: use monotonic buffer for the container also! → yay, time savings!
  - it just disappears on the wink of hand: no destructors called!
  - allocator only needs to bump down a pointer for the buffer!
- Of course: caution!
  - ➢ No side effects in destructors !!!

Wink Out - Example

```
 {
    std::pmr::unsynchronized_pool_resource pool_resrc;
    std::pmr::polymorphic_allocator<std::byte> pool_alloc(&pool_resrc);

    // memory leak?
    auto& data = *pool_alloc.new_object<std::pmr::vector<std::pmr::list<std::pmr::string>>>();

    // use data:
    data.push_back({});
    data.push_back({});

    data[0].push_back("string XXX");
    data[0].push_back("string YYY");

    data[1].push_back("string ZZZ");
}

// out of scope, data gets ‘winked out’ !
// PLUS: no need to call pool_resrc.delete_object(&data) !!! -> deallocation almost free!

// note: requires C++20 for new_object(), C++17 version more verbose
```

- Quote: *“You just destroy an arena and everything winks out!”*

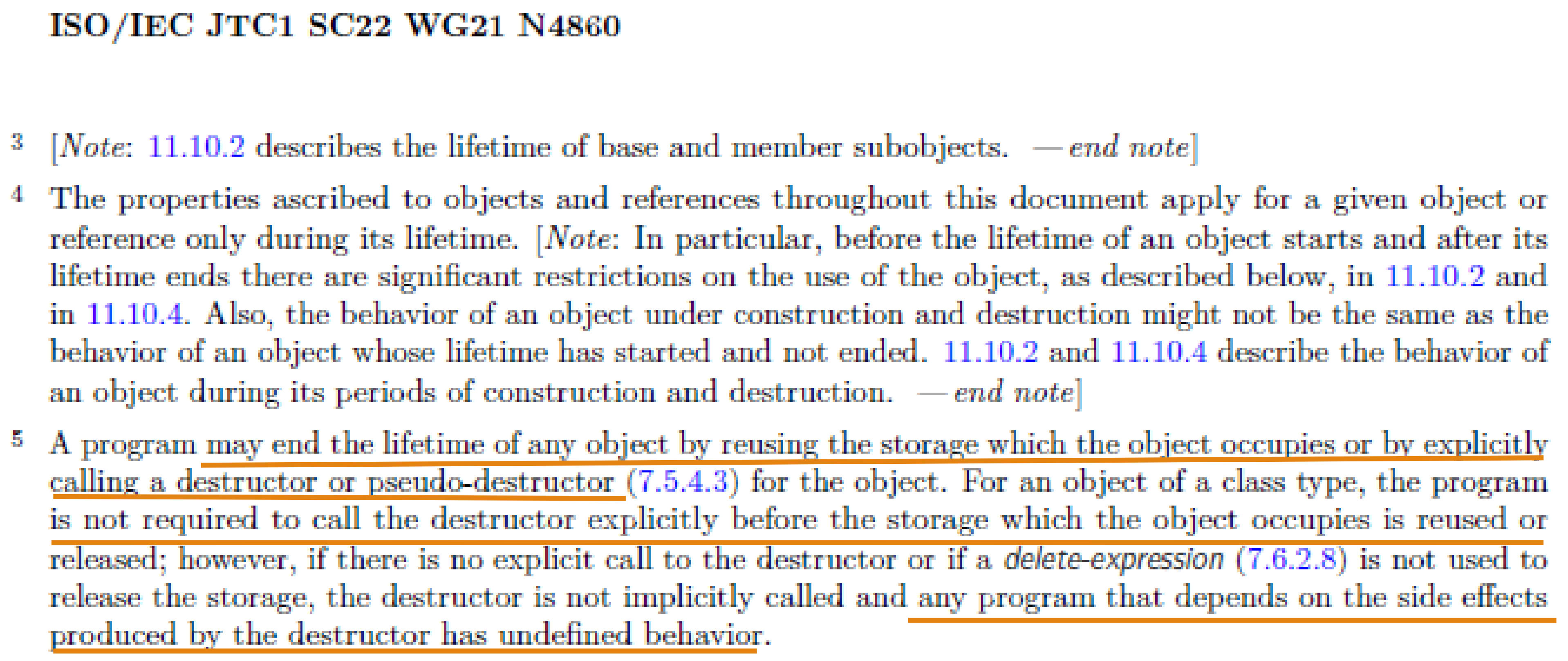
Wink Out - creating objects

```
// naive (Bloomberg style) – crash!
auto data = *new(&pool_resrc) std::pmr::vector<std::pmr::list<std::pmr::string>>(&pool_resrc);


// C++17
using VecListStrg = std::pmr::vector<std::pmr::list<std::pmr::string>>;


void* p = pool_alloc.allocate(sizeof(VecListStrg));
auto vec = static_cast<VecListStrg*>(p);

pool_alloc.construct<VecListStrg>(vec /*, &pool_resrc*/); // ??? PITA!

auto& data = *vec;


// C++20
auto& data = *pool_alloc.new_object<std::pmr::vector<std::pmr::list<std::pmr::string>>>();
```

polymorphic_allocator(
  std::pmr::memory_resource* r);

// GLIBCXX_RESOLVE_LIB_DEFECTS
2969. polymorphic_allocator::construct()
shouldn't pass resource()

Wink Out - is it an UB?



ISO/IEC JTC1 SC22 WG21 N4860

3 [*Note*: 11.10.2 describes the lifetime of base and member subobjects. — *end note*]

4 The properties ascribed to objects and references throughout this document apply for a given object or reference only during its lifetime. [*Note*: In particular, before the lifetime of an object starts and after its lifetime ends there are significant restrictions on the use of the object, as described below, in 11.10.2 and in 11.10.4. Also, the behavior of an object under construction and destruction might not be the same as the behavior of an object whose lifetime has started and not ended. 11.10.2 and 11.10.4 describe the behavior of an object during its periods of construction and destruction. — *end note*]

5 A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling a destructor or pseudo-destructor (7.5.4.3) for the object. For an object of a class type, the program is not required to call the destructor explicitly before the storage which the object occupies is reused or released; however, if there is no explicit call to the destructor or if a *delete-expression* (7.6.2.8) is not used to release the storage, the destructor is not implicitly called and any program that depends on the side effects produced by the destructor has undefined behavior.

Example from a real code base

- C++ Actor Framework (caf)
- JSON data representation
- Tree structure

```
// This JSON abstraction is designed to allocate its entire state in a monotonic
// buffer resource. This minimizes memory allocations and also enables us to
// "wink out" the entire JSON object by simply reclaiming the memory without
// having to call a single destructor. The API is not optimized for convenience
// or safety, since the only place we use this API is the json_reader.

namespace caf::detail::json {
  struct null_t {};

  class value {
   public:
    using array_allocator = monotonic_buffer_resource::allocator<value>;
    using array = std::vector<value, array_allocator>;

    struct member {
      string_view key;
      value* val = nullptr;
    };

    using member_allocator = monotonic_buffer_resource::allocator<member>;
    using object = std::vector<member, member_allocator>;

    using data_type
        = std::variant<null_t, int64_t, double, bool, string_view, array, object>;

    ….
```

Note:
  in CAF *caf::detail::monotonic_buffer_resource* is used because of older platforms

Wink Out in Protocol Buffers

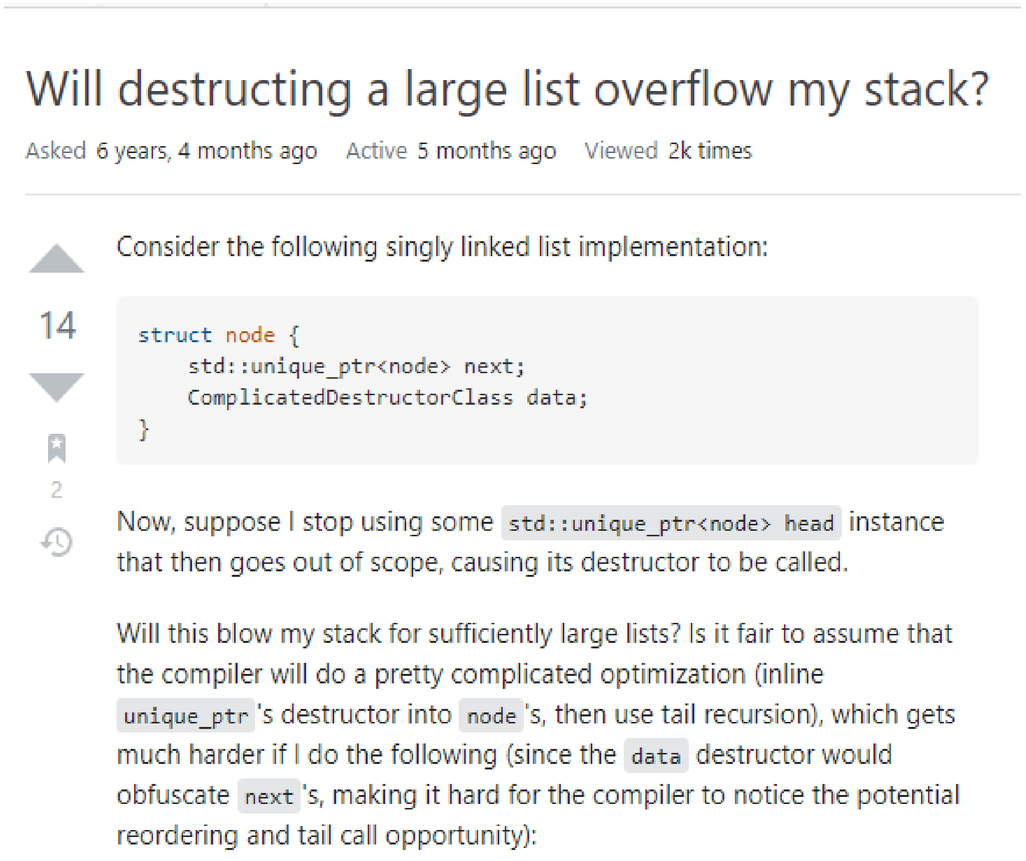
Google‘s *protobuf* for C++ can enable an *Arena* allocator

- Uses a large piece of preallocated memory (called arena)
- We can wink out the whole serialized/deserialized data structure!
  - When the Arena gets destroyed or goes out of scope
  - However – *string* fields are allocated on the heap (AFAIK)
- Additionally, we can also register destructors for some objects!
  - For external and internal objects!
  - *Reset()* will reset the Arena to be used again
- Optimized for the multithreaded use-case
  - "arena-per-request" model works well

Localized GC

Localized garbage collection

- The Problem:
  - Look at that SO ☺ question →
  - But we just try to avoid naked pointers, that surely can‘t be bad!
  - Unbounded recursion/stack depth ☹
- Same problem also in graphs
  - even worse, as there may be also cycles in graphs!
- Herb‘s Sutter talk CppCon 2016:
  - proposed deferred pointers /deffered heap as langauge extension
  - WTF? Do we need it? (spoiler: maybe!)



Will destructing a large list overflow my stack?

Asked 6 years, 4 months ago Active 5 months ago Viewed 2k times

14

2

Consider the following singly linked list implementation:

```
struct node {
    std::unique_ptr<node> next;
    ComplicatedDestructorClass data;
}
```

Now, suppose I stop using some `std::unique_ptr<node> head` instance that then goes out of scope, causing its destructor to be called.

Will this blow my stack for sufficiently large lists? Is it fair to assume that the compiler will do a pretty complicated optimization (inline `unique_ptr`'s destructor into `node`'s, then use tail recursion), which gets much harder if I do the following (since the `data` destructor would obfuscate `next`'s, making it hard for the compiler to notice the potential reordering and tail call opportunity):

Localized garbage collection: 2

- Also called „self-contained heap“!
- Just use the previous Wink Out technique to avoid stack overflow!
- Also here we need C++20 for new_object()

```
struct GraphNode
{
  std::pmr::string m_payload;
  std::pmr::vector<GraphNode*> m_outgoingEdges;

  GraphNode(const std::pmr::string& payload,
            std::pmr::memory_resource* alloc)
    : m_payload(payload, alloc),
      m_outgoingEdges(alloc)
  {
     m_outgoingEdges.reserve(2); // typical fan-out is e.g. 2...
  }
  ~GraphNode() { } // empty!
};


{
  std::array<unsigned, 2*1024> buffer{};
  std::pmr::monotonic_buffer_resource
      mem_resrc(buffer.data(), buffer.size() * sizeof(unsigned));
  std::pmr::polymorphic_allocator<> buff_alloc(&buff_resrc); // C++ 20!

  GraphNode* start = buff_alloc.new_object<GraphNode>("start", &buff_resrc);
  // ...
  GraphNode*    nX = buff_alloc.new_object<GraphNode>(“nodeX", &buff_resrc);

  // cycles are no problem now !!!
  start->m_outgoingEdges.push_back(nX);
  nX->m_outgoingEdges.push_back(start);

  // ‘buff_resrc' destructor releases all graph’s memory here!!!
}
```

Localized GC – creating objects

```
// naive (Bloomberg style) – crash!
GraphNode* start = new(&mem_resrc) GraphNode("start", &mem_resrc);

// C++17
GraphNode* start = static_cast<GraphNode*>((void*)buff_alloc.allocate(sizeof(GraphNode)));

buff_alloc.construct<GraphNode>(start, "start", &mem_resrc);


// C++20
GraphNode* start = buff_alloc.new_object<GraphNode>("start", &mem_resrc);


// -> allocator (AA) aware graph

struct AAGraphNode {
  using allocator_type = std::pmr::polymorphic_allocator<char>;

  explicit AAGraphNode(const std::pmr::string& payload, const allocator_type& alloc);
  std::pmr::string m_payload;
  ...
};

AAGraphNode* start = buff_alloc.new_object<AAGraphNode>("start"); // AA was recognized!
```

std::uses_allocator<> checks it !

Localized garbage collection - wrap-up

- Essentially: Wink Out with naked pointers
- Same caveats as with Wink Out !!!
- Can we use modern C++ here? I.e. smart pointers?
  - Not really, they do not mix well with some PMR usages
  - John Lakos says*: [usage of allocators …]

• Limits applicability of certain standard facilities that manage object lifetimes as they ***neither track nor extend allocator lifetimes****.*
– E.g., std::shared_ptr and std::weak_ptr

* *John Lakos, in "Value Proposition: Allocator-Aware (AA) Software", ACCU 2019*

PMRs and smart pointers

We can use PMR for *shared_ptr*'s internal structures:

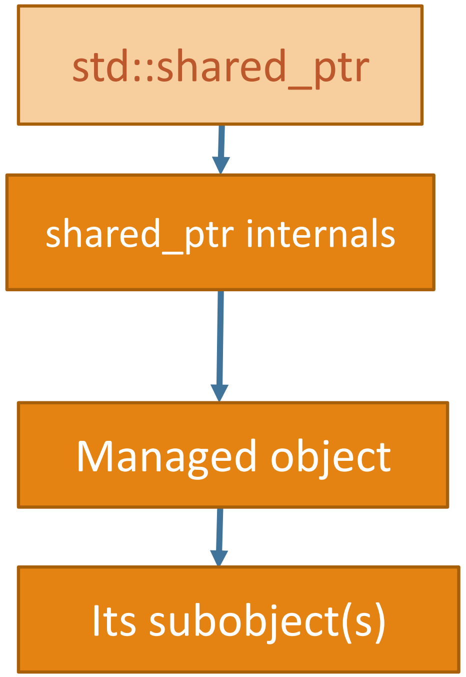
```
template< class Y, class Deleter, class Alloc >
    shared_ptr( Y* ptr, Deleter d, Alloc alloc );
```

We can also create the complete *shared_ptr* using a PMR:

```
template< class T, class Alloc, class... Args >
    shared_ptr<T> allocate_shared(const Alloc& alloc, Args&&... Args);
```

Advisable – allocator should have global scope!

<u>Caution</u>: *shared_ptr* itself is not allocator-aware! :-o
 - no extended copy/move constructors
 - no *get_allocator()* method!
 - thus a container of smart pointers won't forward its allocator into its elements!



C++20‘s destroying delete

void T::operator delete(T* ptr, std::destroying_delete_t); (27) (since C++20)

*27-30) If defined, delete-expressions does not execute the destructor for *p before placing a call to operator delete. Instead, direct invocation of the destructor such as by p->~T(); becomes the responsibility of this user-defined operator delete.*

Use *std::shared_ptr* in a graph?

- What if we want to change parts of the graph and reuse that parts' memory?
- Idea (don't know if it could work):
  1. Use *shared_ptr* to connect graph nodes
  2. Use a pool allocator for the nodes so the blocks are returned to the allocator
  3. Use a *destroying_delete* destructor at the root node to enable Wink Out of the entire graph
- Thus
  1. When deleting a non-root node, *shared_ptr* will delete the sub-nodes
  2. When deleting the entire graph we avoid a stack overflow!

Could that work?
→ What about nodes directly under the root node?
→ Cycles still not allowed ☹

Summing Up

Conclusion - Allocators

- Writing custom data structs/allocators – inherently costly
  - but every developer can just use *std::* ones!
  - predefined pool and arena PMRs
- Not only improving performance, but also:
  - placing objects on the stack / in file mapped memory
  - measuring / reporting memory usage
  - testing correctness
  - implement efficient GC, e.g. in graph structures
- Costs: are ther any?
  - Special-purpose allocators do require training
  - Harder to test
  - *"Winking out" is inherently for experts only** - extreme care required!
  - Gotchas: container shouldn't outlive its allocator, *shared_ptr* not AA, etc...

* *John Lakos, in "Value Proposition: Allocator-Aware (AA) Software", ACCU 2019*

Can we do better?

- We removed the type dependency with PMRs
  - With that we‘ve reached the limits of library approach
- But there are still problems:
  - AA classes require non-trivial constructors
  - Compiler-generated copy operations won’t work
  - Same for C++11 *move* variants
- Towards language proposal … C++23 ???
  - ➢ *„Getting Allocators out of our way“* – Alisdair Meredit, Pablo Halpern

```
// future syntax?
MyHashMap<std::string, std::list<AAA>> hMap using myAllocator;
```

- Push all the plumbing down to the compiler!

Thank you!

Any questions?

In the future…

- C++23 – debug/test allocators (Test resource proposal - P1160 R0)

  “pmr::test_resource is a C++17 memory resource designed for testing that can be plugged into any test framework. It is the modernized version of the bslma::TestAllocator ***used in production for over two decades*** *at Bloomberg, where it has* ***helped to expose a variety of bugs****, such as memory leaks, overruns, multiple deletes, exception-safety guarantee failures etc. ”**

- Use test allocators for performance optimization:
  - *Monotonic_allocator ma;* -> *Counting_allocator ca;* -> *Monotonic_fixed_allocator mf;*
  - allocate buffer by *new()* -> get max. buffer size -> allocate fixed buffer on the stack -> yay, perf. gains!

___

* *CppCon 2019 Talk: “test_resource: The pmr Detective” by Attila Fehér*

Test resource proposal (P1160 R0)

- *std::pmr::default_resource_guard*
  - Install and reset new default memory resource
- *std::pmr::exception_test_loop*
  - Start with no resources, end when *pmr::test_resource_exception* is no more thrown
- *std::pmr::test_resource* – the star! ☆
  - Detect leaks, double frees, buffer overruns
  - Fail to allocate after some limit reached
  - Provide statistics on allocations
- *std::pmr::test_resource_exception*
  - Derived from *std::bad_alloc*
- *std::pmr::test_resource_monitor*
  - Observes changes in *pmr::test_resource_exception*'s statistics

C++89/03 Allocator problems: 1

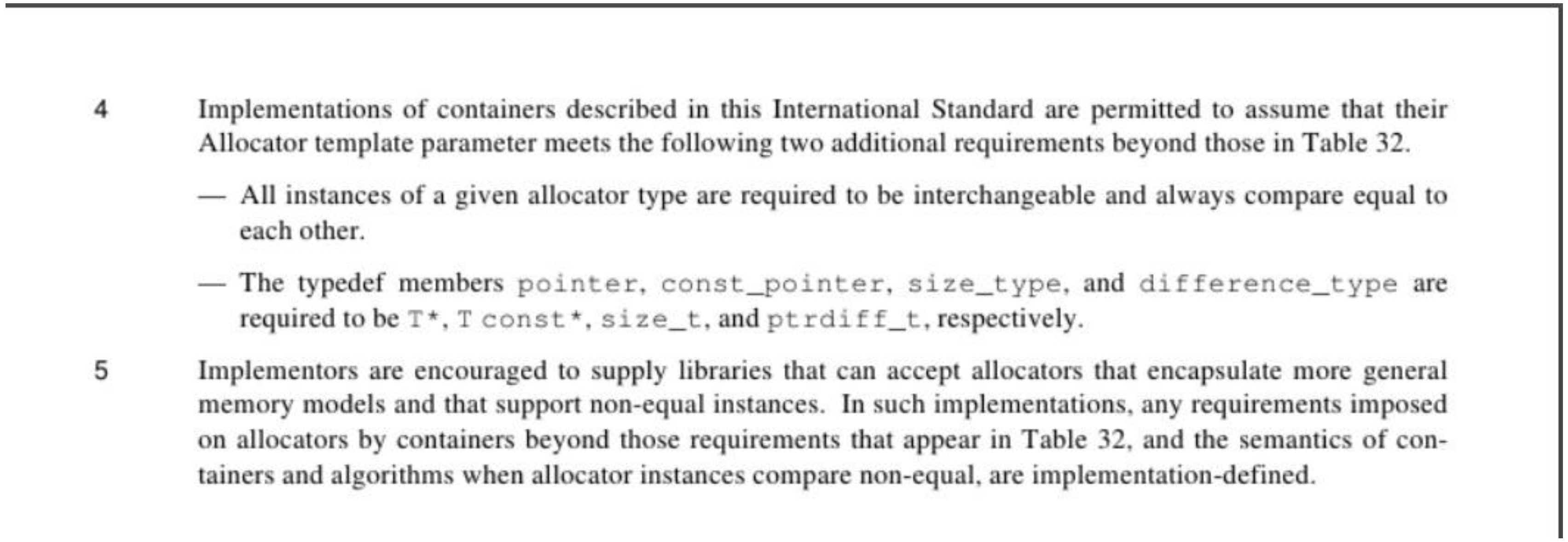
The original design was technically lacking in several aspects

- Because of time pressure (?)

1. *std::allocator* instances are stateless and so they always compare equal
   - Allocator equality implies interchangeability i.e. the instances can free each other's memory!
2. *pointer'*s typedef is always *T*,* i.e. no fancy pointers!
3. What allocators are used in container of containers is sometimes not specified!!!
   - depending on capacity at the time of insertion either assignment or copy construction will be used
   - … and they will use different allocators in both cases
   - Not what we wanted, no control over allocations, more kinda mess!

C++89/03 Allocator problems: 2

- *std::allocator* instances are stateless and so always compare equal
- Allocator equality implies interchangeability i.e. the instances can free each other's memory!
- *pointer* typedef is always *T*,* i.e. no fancy pointers!
- because:



4 Implementations of containers described in this International Standard are permitted to assume that their Allocator template parameter meets the following two additional requirements beyond those in Table 32.

— All instances of a given allocator type are required to be interchangeable and always compare equal to each other.

— The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `size_t`, and `ptrdiff_t`, respectively.

5 Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 32, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

C++89/03 Allocator problems: 3

- C++ 98 Standard says (containers requrements section):

  assignment operation

  ```
  cont1 = cont2;
  ```

  does not copy *cont2*'s allocator to *cont1*

  → but…

  the copy constructor

  ```
  T cont3(cont2);
  ```

  does copy *cont2*'s allocator to *cont3*

- Inconsitency, what allocators are used in container of containers is sometimes not specified!!!
  - depending on capacity at the time of insertion either assignment or copy construction will be used
- Not what we wanted, no control over allocations, more kinda mess!

C++17 monotonic buffer resource

```
unsigned buff[1024] = {};

std::pmr::monotonic_buffer_resource arr_mem_res(buff, sizeof(buff));


std::pmr::vector<std::string> vec(&arr_mem_res);


// OR: template typedef
template <class T> using
    my_pmr_vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;

my_pmr_vector<std::string> pmrvec(&arr_mem_res);
```

Erased Allocators (C++ 11)

- *std::function, std::promise, std::future …* in C++11
  - type erasure in C++: don‘t use *T* in class type, use a template method and a wrapper instead

```
class Object {
 template<typename T> struct ObjectModel: ObjectConcept
 {
     ObjectModel(const T& t) : object(t) {}
     virtual ~ObjectModel() {}
  private:
     T object;
 };

 std::shared_ptr<ObjectConcept> m_object;

 public:
  template<typename T> Object(const T& obj)
     : m_object(new ObjectModel<T>(obj)) { }
```

- NOT chosen for memory resources!