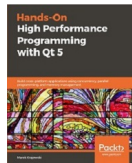# PMRs for Performance in C++17-20

MAREK KRAJEWSKI

whoami()

- C++ (et al.)

- Networking protocols

- Client-Server

- Qt and UI

- Linux, Windows, Embedded

- Freelancer

- Concerned about performance

Why this talk?

- I was listening to *C++ Weekly's Ep. 250**
  - advanced PMR techniques mentioned (*wink-out, local GC*) , but I didn't reall get it
  - started looking into it and…

- What I found:
  - the surprisingly lacking design of C++98 allocators
  - a fascinating story of library design and evolution
  - some techniques which aren't generally known

- What this talk isn't:
  - expert's talk
  - tutorial on writing allocators (legacy and new-style)
  - tutorial on writing allocator aware (AA) classes
  - live performance comparision session on https://www.quick-bench.com/

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

Overview

- Memory Allocators for Performance

- Allocators in C++98 and C++11

- Allocators in C++17 and PMRs

- Usage Examples

- Advanced PMR Techniques

- Allocators in C++20 (and Beyond)

Memory and performance

- Allocating Memory

- Accessing Memory
  - cache hierarchy
  - cache invalidation (& importance of locality)

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

- Custom Memory Allocators

  → Let's keep that talk on intermediate level!

Why custom memory allocators?

1. For performance!
   i. Allocate related objects in contiguous memory
      - for locality
      - for separation (e.g. hot/cold data separation)
   ii. Avoid locks when in the same thread
      - however: tcmalloc / jemalloc / gcc malloc already use thread-local memory buffers!!!
   iii. Fight fragmentation / diffusion
   iv. Lower costs of allocations

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

Traditional allocator wisdom

- Sized blocks allocators
  - e.g. for network packets

- C++ allocator overloading: global / in class
  - overload the *new()* operators, STL *Allocator<T>*

- Game programming community: the classic [*Book 1*]
  - stack(ed) allocators – memory <u>cannot</u> be freed in arbitrary order!
  - 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 <u>resetted</u>!

- Serialization
  - Google's Protocol Buffers - *protobuf::Arena* class

- XML parsing using lists for DOM representation [*Book 2*]
  - stacked allocator, but can release elements
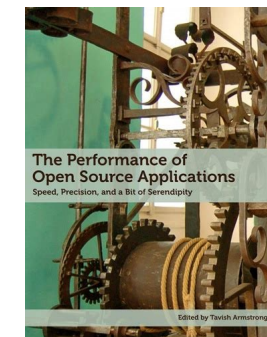
Custom Containers vs Custom Allocators

"

Programmers often believe they need to write custom containers to get this optimization instead of using the standard containers such as *std::vector<T>* and *std::list<T>*. This is certainly doable. However I do not believe this is the best way to go.

Instead I prefer to write a custom allocator which can be used with any of the standard containers.

HOWARD HINNANT, 2015

"

Overloading C++'s new() operator

```cpp
// override global memory handling

void* operator new(size_t size) { ...; }
void operator delete(void* p) { ... }

void* operator new[](size_t size) { ...; }
void operator delete[](void* p) { ... }


// or only for a single class!

class Person
{
public:
    void* operator new(size_t size) { ...; }
    void operator delete(void* p) { ... }

    // etc...

};
```

Technical note: The preceding example shows C++11 code. C++17 adds overloads to specify alignment. C++98 used the now deprecated *throw()* specifier.

STL Allocators

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

```cpp
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*
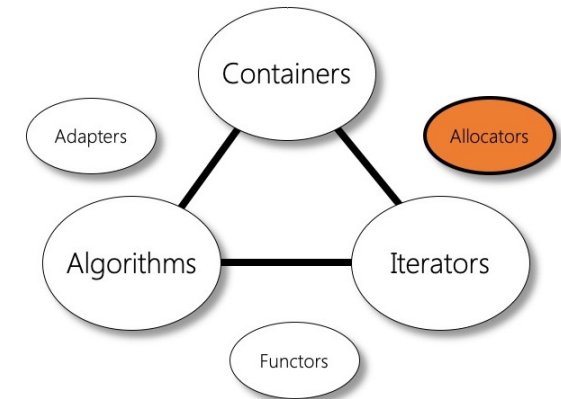
C++98/03 Allocator API 1

- Several nested types in std::allocator<T>

```cpp
template<typename T> class allocator
{
public:
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef T value_type;

    template <class U> struct rebind { typedef allocator<U> other; }

    // ...
};
```

- Rebind mechanism

```cpp
typedef typename allocator::template rebind<list_node<T>>::other node_allocator;
```

C++98/03 Allocator API 2

- The functional API:

```cpp
template<typename T>
    class allocator
{
public:

    // ...

    template <class U> struct rebind { typedef allocator<U> other; }

    pointer allocate(size_type n, allocator<void>::const_pointer hint = 0);
    void deallocate(pointer p, size_type n);

    void construct(pointer p, T const& val);
    void destroy(pointer p);

};


// comparison operators:
template<class T, class U> bool operator ==(allocator<T> const&, allocator<U> const&);
template<class T, class U> bool operator !=(allocator<T> const&, allocator<U> const&);
```

C++89/03 Allocator problems: 1

- It's a part of container's type!

```cpp
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 , not scalable...

- However
  - this could be a good thing if pointers were of different size/type!
  - as in segmented memory adressing!

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++98 Allocators – our wishes

- The allocator used to construct a container should also be used to construct the elements of that container.
  - Note: this principle eventually became known as the scoped allocator model (!)

- Stateful allocators and fancy pointers should be supported
  - e.g. for pointers into a shared memory segment!

- An object's type should be independent of the allocator it uses to obtain memory.
  - as it is only an implementation detail
  - we also don't care if the object is on the stack, heap or in register, has local or global linkage, etc…

Bloomberg Design: 1

- Before STL/C++98:
  - They had developed a set of stateful, generally non-equal allocators before STL (aka *Lakos allocator model*)
  - Not part of container's type! - passed by pointer to each allocator-aware (AA) class
  - They had to merge both worlds as not to give up on the advantages of the STL!
  - Finally, they joined the committee to work on allocators

- 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
    - if not used, fallback to the default new/delete allocator - i.e. STL classes won't see a difference!

  - 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 > > >* !!!

Allocators, C++11

- What have we got in C++11?

- Support for stateful allocators
  - i.e. wording explicitely allows it – the old allowable assumptions were removed!

- Support for fancy pointers (through *std::pointer_traits*)

- Simplified allocator requirements:
  - some requirements are now optional!
  - The template *std::allocator_traits* supplies the default implementations for all optional requirements
  - standard library containers and other allocator-aware classes access the allocator through *std::allocator_traits*, not directly!
  - i.e. we can write allocators without much of a hassle → yay, simplicity!

Example simple C++11 allocator

```cpp
template <class T> struct TestAllocator
{
    typedef T value_type;

    template <class U> constexpr TestAllocator(const TestAllocator<U>&) noexcept {}

    T* allocate(std::size_t n)
    {
        if (n > std::size_t(-1) / sizeof(T))
            throw std::bad_alloc();

        if (auto p = static_cast<T*>(std::malloc(n * sizeof(T))))
            // TRACE("allocated");
            return p;

        throw std::bad_alloc();
    }

    void deallocate(T* p, std::size_t) noexcept { std::free(p); }
};

template<class T, class U> bool operator==(const TestAllocator<T>&, const TestAllocator<U>&) { return true; }
template<class T, class U> bool operator!=(const TestAllocator<T>&, const TestAllocator<U>&) { return false;}
```

Example C++ 11 allocator traits

```cpp
// use TestAllocator

std::vector<int, TestAllocator<int>> vec;

vec.push_back(1);
vec.push_back(2);

// use traits
std::allocator_traits<TestAllocator>::pointer p;
std::allocator_traits<TestAllocator>::size_type sz;

if (std::allocator_traits<TestAllocator>::is_always_equal())
{
    // stateless allocator!
}


// etc..


// use allocator „directly"
std::allocator_traits<TestAllocator> at;

at.construct();
at.destroy();
at.max_size();
```

Allocators, C++11 contd.

- But what about the Bloomberg design?

- Allocator propagation:
  - C++11 added scoped allocator support!
  - *std::scoped_allocator_adaptor<Alloc>* template - implements multi-level allocator for multi-level containers, adapts legacy allocators
  - *std::uses_allocator* trait - checks if the specified type supports uses-allocator construction

- But also added new required trait types in *std::allocator_traits* to control allocator propagation
  - i.e. *propagate_on_container_copy_assignment, propagate_on_container_move_assignment, propagate_on_container_swap*
  - aka POCCA, POCMA and POCS
  - defaults never propagate!

- And a new function in *std::allocator_traits - select_on_container_copy_construction()*
  - also needed for propagation control

C++ 11 Allocators: short_alloc

```cpp
// Create a vector<T> template with a small buffer of 200 bytes.
// Note for vector it is possible to reduce the alignment requirements
// down to alignof(T) because vector doesn't allocate anything but T's.
// And if we're wrong about that guess, it is a compile-time error, not
// a run time error.

template <class T, std::size_t BufSize = 200>
    using SmallVector = std::vector<T, short_alloc<T, BufSize, alignof(T)>>;

// Create the stack-based arena from which to allocate
SmallVector<int>::allocator_type::arena_type a;


// Create the vector which uses that arena.
SmallVector<int> v{ a };

// Exercise the vector and note that new/delete are not getting called.
v.push_back(1);
v.push_back(2);

// ...


// allocator + vector go out of scope
```
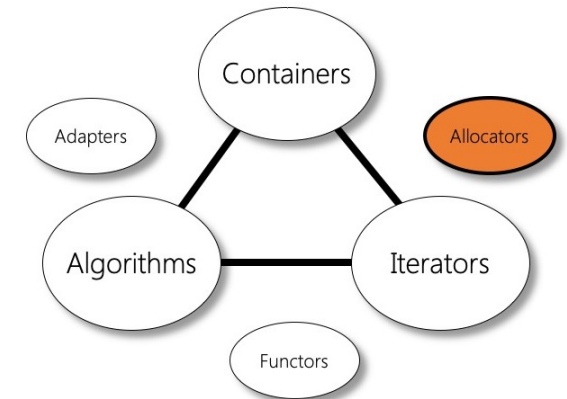
C++ Allocators, redux

- OK, we got some changes in C++11

  → 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 are still part of type signature

- Remember?
  a) WTF?
  b) PITA!
  c) Etc…

- OK: In C++11 some new classes got a „type erased" allocators
  - *std::function , std::promise, std::shared_ptr*
  - But not the containers!

Bloomberg's design, part 2

- Remember Bloomberg's design part 1?

- **wrap** the base class for allocators in an STL-conformant *Allocator* wrapper
- then always **use the wrapper** in the signatures of the STL containers
  - ➢ if not used, fallback to the default new/delete allocator - i.e. STL classes won't see a difference!

→ So now, just let us use that design in C++17 to remove the dependency on allocators!

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>

C++17 polymorfic memory resources

std::container

<std::pmr::polymorfic_allocator>

<>
std::pmr::memory_resource

It isn't called allocator anymore!

std::pmr::monotonic_buffer_resource

My Specific Memory Resource
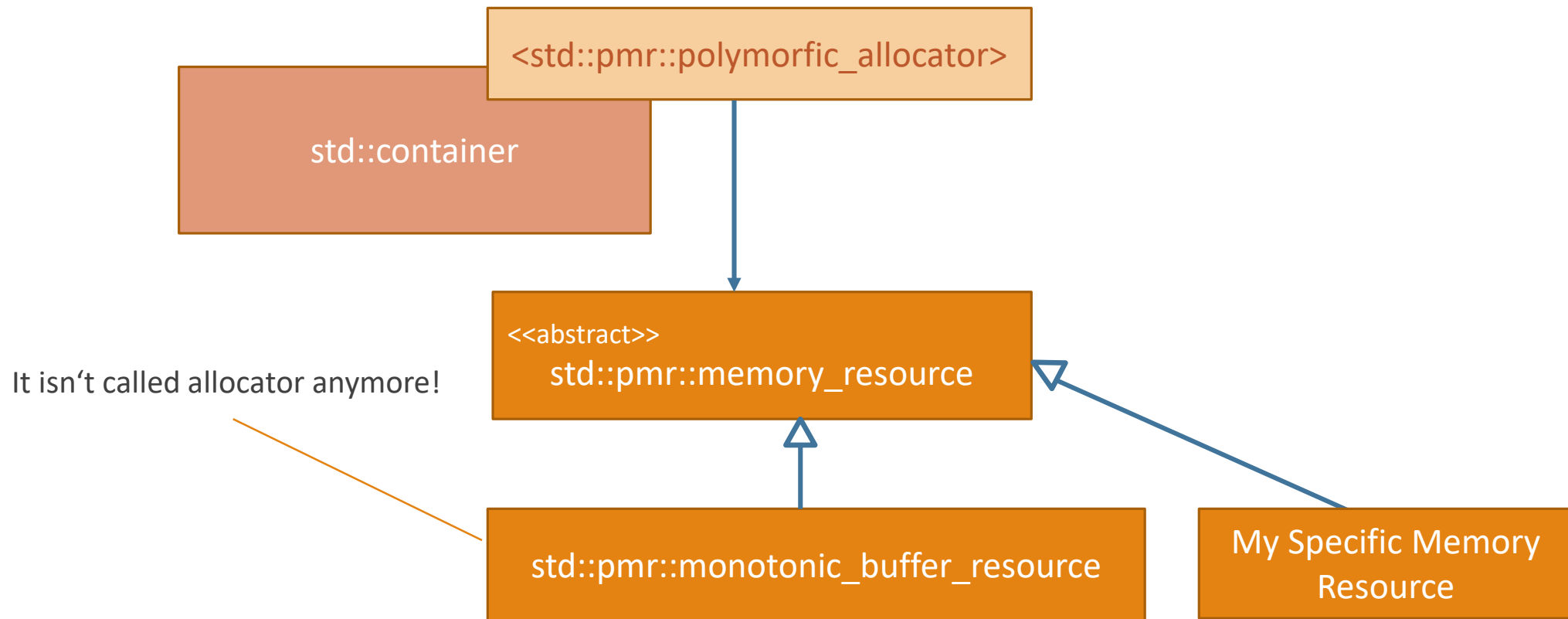
C++17 polymorfic memory resources

```cpp
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);
```

PMR Usage Examples

Available PMR types

- *std::pmr::monotonic_buffer_resource*
  - releases the allocated memory only when the resource is destroyed
  - stacked allocator, on-stack allocator, single-frame allocator
  - In maths talk: monotonic = always growing, i.e. elements rarely deleted!

- *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 (good locality)

- *std::synchronized_pool_resource*
  - i.e. pool allocator with locks

- *std::pmr::pool_options*
  - can parametrize the pool resource

Usage of PMR's

- Basic usage

```cpp
std::pmr::vector<std::pmr::string> vec1(&buffer_mem_res);
std::pmr::vector<int> vec2(&buffer_mem_res);
```

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

```cpp
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
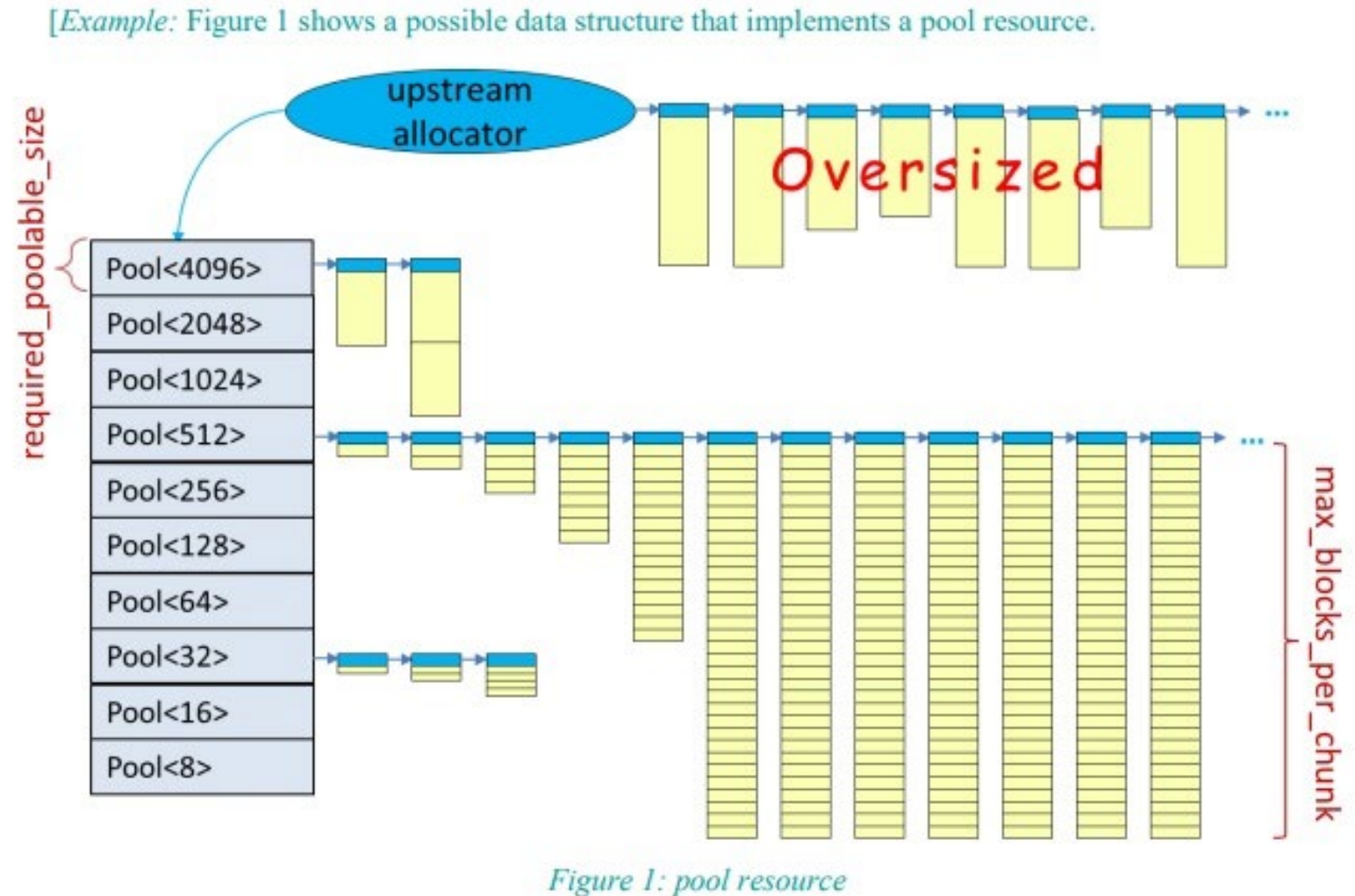
– end example]

Upstream allocators

- „Upstream" allocators ?
  - Of course not! They are upstream memory resources!!!

- W 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 *monotonic_buffer_resource* as an upstream for *unsynchronized_pool_resource*

- Default upstream:
  - *std::pmr::new_delete_resource()*

Setting the default allocator

- Bloomberg - setting the default and global allocators

- In C++17: only the default resource
  - *std::pmr::set_default_resource(std::pmr::memory_resource* r)*
  - *std::pmr::get_default_resource()*

- What default?
  - *std::pmr::new_delete_resource() -* uses the global operators *new()* and *delete()* to allocate and deallocate memory
  - *std::pmr::null_memory_resource -* function returning a static *memory_resource* that performs no allocation

Advanced Techniques

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 constructors called!
  - allocator only needs to bump down a pointer for the buffer!

- Of course: caution!
  - ➢ No side effects on destuctors !!!

Wink Out - Example

```cpp
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!"*

Localized garbage collection: 1

- 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

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 use C++20 for new_object()

```cpp
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 garbage collection: 3

- Same caveats as with wink-out !!!

- What if we need to deallocate a node?
  - As in the XML parsing example…

- I didn't test it with *std::unique_pointer* (*std::shared_pointer* if cycles?)
  - Sorry, lack of time! ☹
  - **More work needed**

EBO and Allocators

- Was the stateless design of C++98 **all** bad?

No – as they were stateless, they could be optimized away by EBO (empty base class optimzation)!
Yay – memory size gains!

→ We can detect if the allocator is stateless and emulate EBO! Gcc stdlib code:

```cpp
/// Specialization using EBO.
template<int _Nm, typename _Tp>
  struct _Sp_ebo_helper<_Nm, _Tp, true> : private _Tp
{
   explicit _Sp_ebo_helper(const _Tp& __tp) : _Tp(__tp) { }
   explicit _Sp_ebo_helper( _Tp&& __tp) : _Tp(std::move(__tp)) { }
   static _Tp& _S_get(_Sp_ebo_helper& __eboh) { return static_cast<_Tp&>(__eboh); }
};

/// Specialization not using EBO.
template<int _Nm, typename _Tp>
  struct _Sp_ebo_helper<_Nm, _Tp, false>
{
   // ...
private:
   _Tp _M_tp;
};
```

C++20 and the future

C++20 Changes

- std::byte
- != removed

C++20 Changes contd.

- C++17 deprecations removed in C++20
- rebind: … ???
- Is always equal: ???
- pointer: ???

C++ 20 Additions

- *std::uses_allocator_construction_args(const Alloc& alloc,   Args&&... Args)*
  - prepares the argument list matching the flavor of uses-allocator construction required by the given type

- *std::make_obj_using_allocator (const Alloc& alloc,   Args&&... Args)*
  - creates an object of the given type by means of uses-allocator construction
  - Equivalent to;

```cpp
return std::make_from_tuple<T>(
    std::uses_allocator_construction_args<T>(alloc, std::forward<Args>(args)...)
    );
```

- *std::uninitialized_construct_using_allocator(T* p, const Alloc& alloc, Args&&... args);*
  - creates an object of the given type at specified memory location by means of uses-allocator construction

Future use cases

- C++23 – debug/test allocators

  "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

What now?

- We've reached the limits of library approach!!!

- Towards language proposal … C++23 ??
  - ➢ *„Getting Allocators out of our way"* – Alisdair Meredit, Pablo Halpern

    ```cpp
    // future syntax?
    MyHashMap<int, int> x using myAllocator;
    ```

Conclusion

- Writing custom data structs/allocators – inherently costly
  - but every developer can just use *std::* ones!
  - predefined pool and arena PMRs

- Not only improved 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

@mrkkrj

Thank you!

Any questions?

Erased Allocators (C++ 11)

- std::function, std::promise, std::future … in C++11

- NOT chosen for memory resources!

- type erasure in C++: don't use T in class type, use a template method and a wrapper instead

```cpp
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 )) { }
```

Own PMR-aware classes

- *std::uses_allocator* construction

```
template <class T, class Alloc >
  struct uses_allocator;
```

Checks if the specified type supports uses-allocator construction:

"If T has a member typedef allocator_type which is convertible from Alloc or is an alias of std::experimental::erased_type, the member constant value is true."

Uses-allocator construction

There are three conventions of passing an allocator alloc to a constructor of some type T:

- if T does not use a compatible allocator (std::uses_allocator_v<T, Alloc> is false), then alloc is ignored.
- otherwise, std::uses_allocator_v<T, Alloc> is true, and
  - if T uses the *leading-allocator convention* (is invocable as `T(std::allocator_arg, alloc, args...)`), then uses-allocator construction uses this form
  - if T uses the *trailing-allocator convention* (is invocable as `T(args..., alloc)`), then uses-allocator construction uses this form
  - otherwise, the program is ill-formed (this means std::uses_allocator_v<T, Alloc> is true, but the type does not follow either of the two allowed conventions)
- As a special case, std::pair is treated as a uses-allocator type even though std::uses_allocator is false for pairs (unlike e.g. std::tuple): see pair-specific overloads of std::polymoprhic_allocator::construct and std::scoped_allocator_adaptor::construct (until C++20) std::uses_allocator_construction_args (since C++20)

The utility functions std::make_obj_using_allocator, and std::uninitialized_construct_using_allocator may be used to explicitly create an object following the above protocol, and std::uses_allocator_construction_args can be used to prepare the argument list that matches the flavor of uses-allocator construction expected by the type. (since C++20)

Specializations

Custom specializations of the type trait **std::uses_allocator** are allowed for types that do not have the member typedef allocator_type but satisfy one of the following two requirements:

1) T has a constructor which takes std::allocator_arg_t as the first argument, and Alloc as the second argument.
2) T has a constructor which takes Alloc as the last argument.