Basic usage of PMRs for better performance

MAREK KRAJEWSKI

whoami()

- Freelancer
- C++ (et al.*)
- *previously*: networking protocols, Client-Server, REST
- *now*: Qt and UI
- Linux, Windows, Embedded
- *always*: concerned about performance

* *C, C++, bash, Python, Tcl, Java, Groovy, Ocaml, Lisp, F#, Javascript, Haskell, Elm*

Overview

- Intro
- What are allocators?
- STL Allocator design (and its flaws)
- What problems PMRs are solving?
- Which PMR is to be used in which scenario?
- Comparison with system allocator tuning (*jemalloc*)
- Wrap-up and questions

What are allocators?

Memory and performance

- Accessing Memory – slow!
  - *“The slowest part of computer hardware in the modern day is not the processor, it’s the memory bus!"*
  - CPU cache hierarchy
  - cache invalidation (& importance of data locality)
- Allocating Memory – costly!
  - syscalls
  - synchronization
  - fragmentation
- Allocators try to solve both of these problems for us!

Two kinds of memory allocators

- System Memory Allocators (*ptmalloc*, *tcmalloc*, *jemalloc, NT Heap, mimalloc, etc*)
  - in *libc / LD_PRELOAD*
  - Invoked by *malloc/free,* set globally for the entire program
  - Generic, very good performance in most situtations, but…
  - *"In order to cover a **wide range of applications**, the default setting is sometimes kept conservative and suboptimal, even for many common workloads."*
- Custom Memory Allocators
  - Invoked explicitly in the program
  - Can be used only in specific parts of program
  - Simpler to customize
  - Also good for: debugging, special placement of data, profiling, etc

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

Custom allocators performance gains

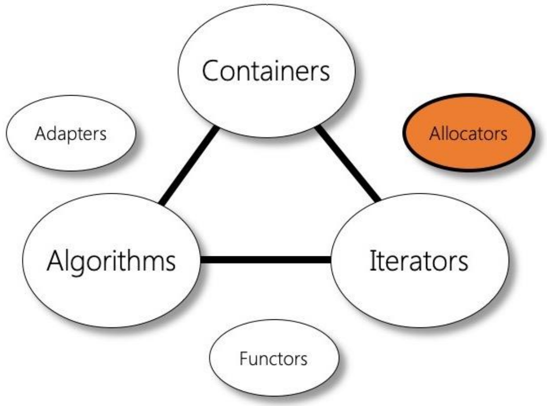
- Performance gains can arise from:
  - Faster allocation/deallocation calls
  - Improved memory access (i.e. data locality)
- Which one dominates?
  - Short-running programs: faster allocation calls
  - Long-running programs: improved memory access

STL Allocator design

… AND ITS FLAWS

Support for custom allocators C++

```
// 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...

};
```



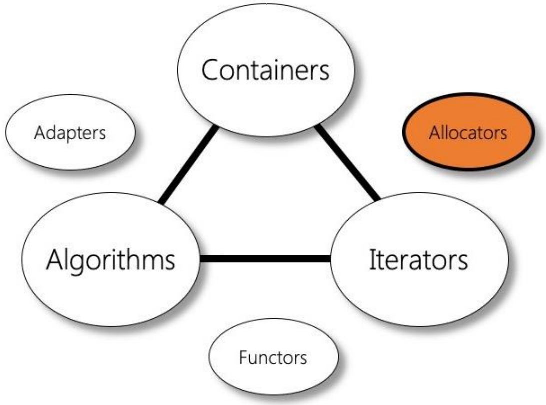

```
// STL containers – allocators as template parameters
template<class T, class Allocator = allocator<T>> class vector;
```

---
* *The examples shows C++11 code. C++17 adds overloads to specify alignment. C++98 used the now deprecated throw() specifier.*

C++98/03 Allocator API 1

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

```
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; }

    ...
};
```



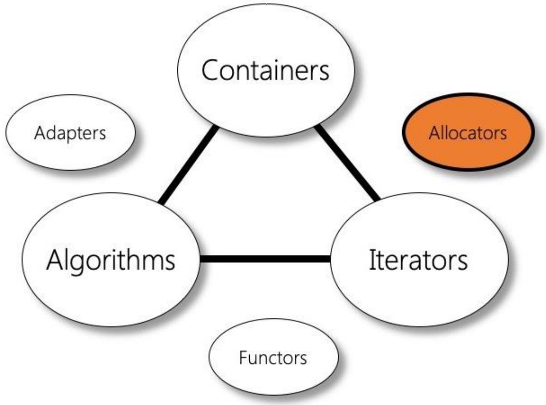
- Rebind mechanism

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

C++98/03 Allocator API 2

- The functional API:

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

    ...

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

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

    // call constr./destr.
    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&);
```



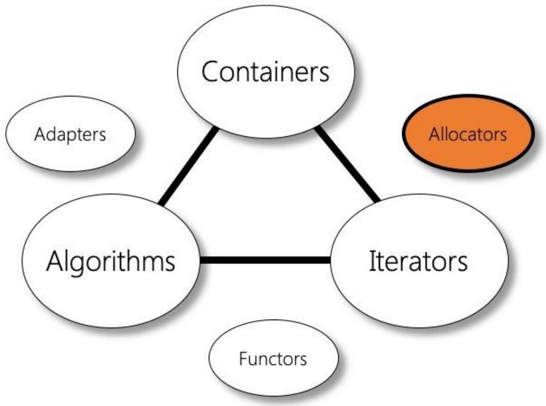
C++98/03 Allocators and Containers

- STL containers have it as template parameter:

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

- Container takes Allocator's *pointer* and *reference* definitions and reuses it
  - ➢ Reason for including it in container's type!

- Instance then stored in container
  - Container calls *allocate()/deallocate()* and *construct()/destroy()*



- Code example (Visual Studio 2022*):

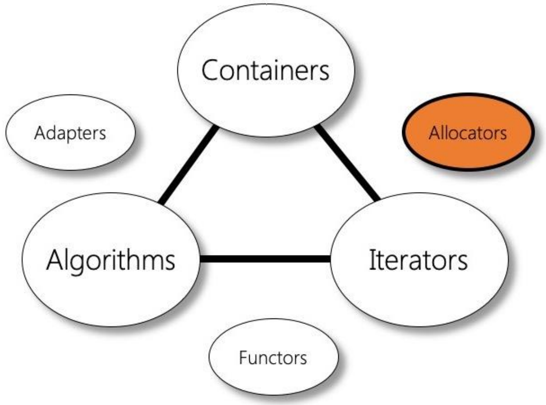
```
_CONSTEXPR20 vector(_CRT_GUARDOVERFLOW const size_type _Count, const _Ty& _Val, const _Alloc& _Al = _Alloc())
    : _Mypair(_One_then_variadic_args_t{}, _Al)
 {
    _Construct_n(_Count, _Val);
 }

_Compressed_pair<_Alty, _Scary_val> _Mypair;  // SCARY → decouples iterator‘s type from the allocator!
```

___
* *gcc uses `_Sp_ebo_helper` class for that*

Allocators in C++11

→ What changed in C++11?

- Support for stateful allocators ✓
- Support for fancy pointers (through *std::pointer_traits*) ✓
- Scoped allocator support (i.e. allocator forwarding) ✓



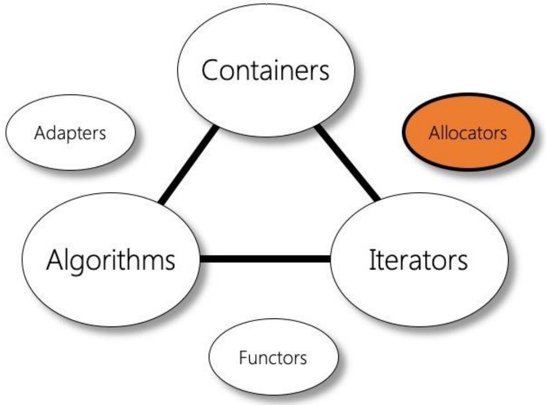
→ What was left for C++17?

- Container's type shouldn't depend on 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…

C++11 Allocator problems

- It‘s 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 !!!
```



- Thus theoretically:

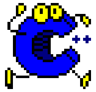
```
// must/could be:
template <class Alloc> void func(const std::vector<int, Alloc>& v);
```

- WTF?* Excuse me, but this doesn‘t scale! PITA!!!
- OK, in C++11 some new classes got „type erased“ allocators
  - *std::function , std::promise, std::shared_ptr,* but not the STL containers!

* *However, this could be a good thing if pointers were of different size/type (as in segmented memory addressing)!*

What problems are PMRs solving?

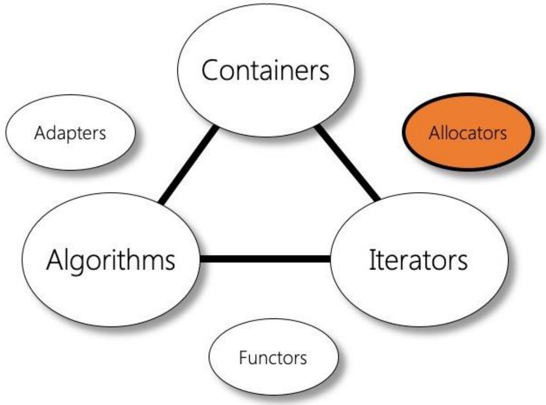
C++17 Allocators, problems to fix

- In C++11 allocators are still part of type signature! 
  - Remember?

  a) WTF?

  b) PITA!

  c) Etc…



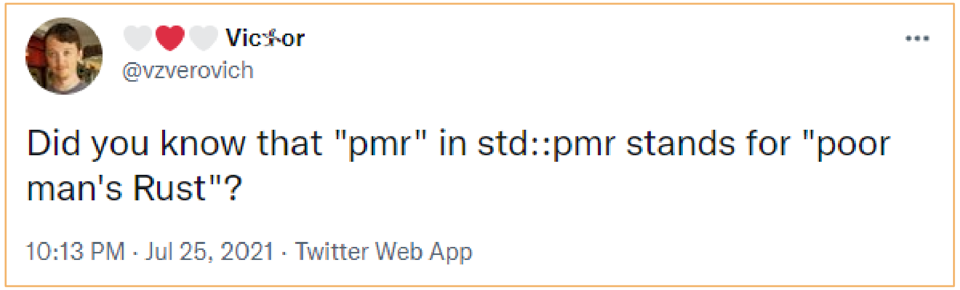
- The C++17 solution for containers:
  - wrap a base class for new allocators in an STL-conformant *Allocator* wrapper
  - then always use the wrapper in the signatures of the STL containers!
  - A single wrapper type uses different PMRs internally (polymorfism, right?)

Enter PMRs!

What is a PMR?

→ poor man‘s Rust? ☺

→ personne mobilité réduite? ☺



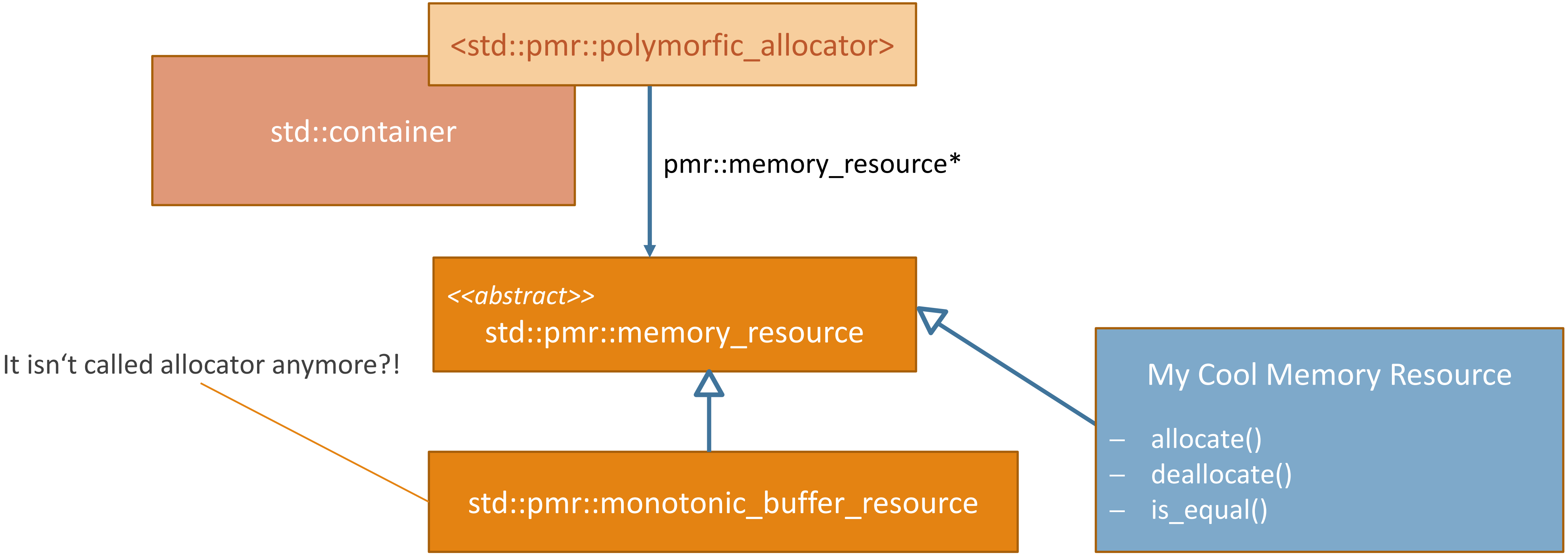
No!!! It‘s:

→ Polymorphic Memory Resource

Wait …

→ why not Polymorphic Allocator?

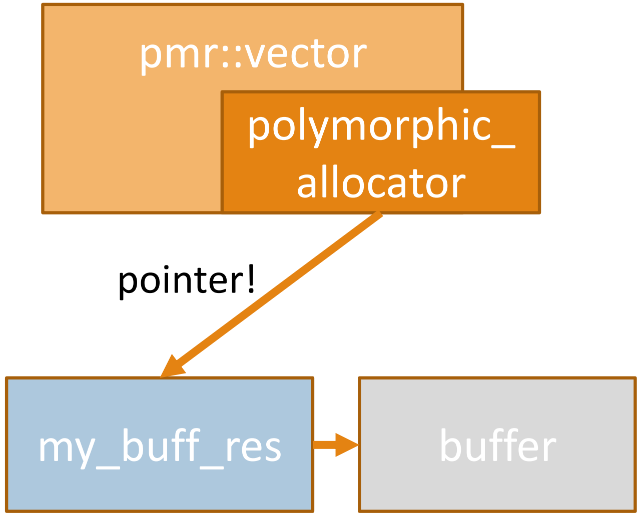
C++17 Polymorfic Memory Resources



C++17 Polymorfic Memory Resources

```cpp
unsigned buffer[1024] = {};
my_cool_buffer_resource my_buff_res(buffer, sizeof(buffer));

std::pmr::vector<std::string> std_pmr_vec(&my_buff_res);

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

pmr_vector<std::string> my_pmr_vec(&my_buff_res);
```

implicit conversion from `std::pmr::memory_resource*` to `std::pmr::polymorphic_allocator`!



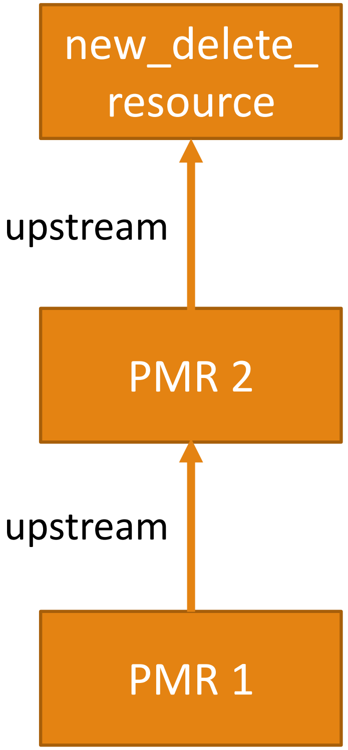
- **Problem 1**: container shouldn’t outlive its memory resource (pointers!!!)
  - Think about function-local static objects!
- **Problem 2**: PMR not forwarded to *std::string* instances!
  - We should write: `std::pmr::vector<std::pmr::string> pmr_vec(&buff_res);`

Upstream allocators/resources

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

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

- If we don't set any, the default upstream will be used:
  - *std::pmr::new_delete_resource()*
- Default allocator/resource:
  - *std::pmr::set_default_resource(std::pmr::memory_resource* r)*



Implementing own PMR classes?

- Implementing own *std::pmr::memory_resource* subclass
  - not that hard, we only need to implement:

```
void* do_allocate(std::size_t bytes, std::size_t alignment);
void do_deallocate(void* ptr, size_t bytes, size_t alignment);
bool do_is_equal(const std::pmr::memory_resource& other);
```

- Implementing own allocator-aware (AA) class
  - like *pmr::string, pmr::vector* etc.
  - More difficult:
    - AA classes require non-trivial constructors
    - Compiler-generated copy operations won't work
    - Same for C++11 *move* variants
  - i.e. constructor, copy, move and assignment operators have to take care of the allocator!
    - we won't discuss it in this talk

PMRs in standard library

Available PMR types

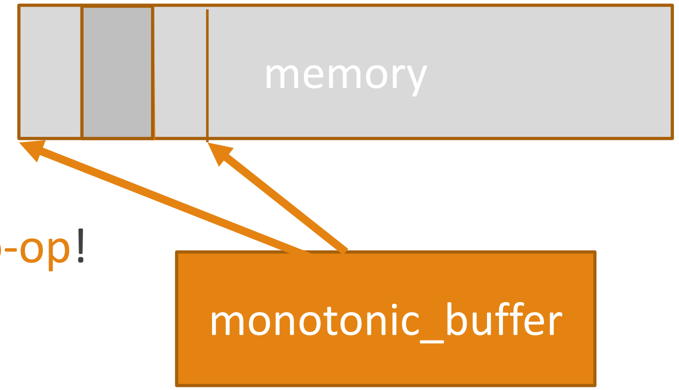
1. two basic PMRs
   - *std::pmr::monotonic_buffer_resource* - In maths talk: monotonic = always growing, i.e. elements deleted only when the resource is destroyed
   - *std::pmr::(un)synchronized_pool_resource* - A pool allocator: it consists of a collection of *pools* that serve requests for different block sizes.

2. two special PMRs
   - *std::pmr::new_delete_resource()* - function returning a static *memory_resource* that uses 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

3. many containers in *std::pmr::* namespace
   - *std::pmr::vector, std::pmr::list, std::pmr::string,* etc…
   - Typedefs for regular STL containers using *std::pmr::polymorphic_allocator* by default!

Basic PMRs 1: Monotonic buffer resource

- Designed for very fast memory allocations
  - just bump-up an internal pointer!
- memory released all at once when resource goes out of scope
- memory increases monotonically, because its *deallocate()* member is a no-op!
  - In maths talk: monotonic = always growing, i.e. elements rarely deleted!
  - i.e. the memory of a freed object remains unavailable (wasted space ▢)!



Constructors:

```
monotonic_buffer_resource(); // uses default resource for upstream
monotonic_buffer_resource(std::pmr::memory_resource* upstream);

monotonic_buffer_resource(std::size_t initial_size);
monotonic_buffer_resource(std::size_t initial_size, std::pmr::memory_resource* upstream);

monotonic_buffer_resource(void* buffer, std::size_t buffer_size);
monotonic_buffer_resource(void* buffer, std::size_t buffer_size, std::pmr::memory_resource* upstrm);
```

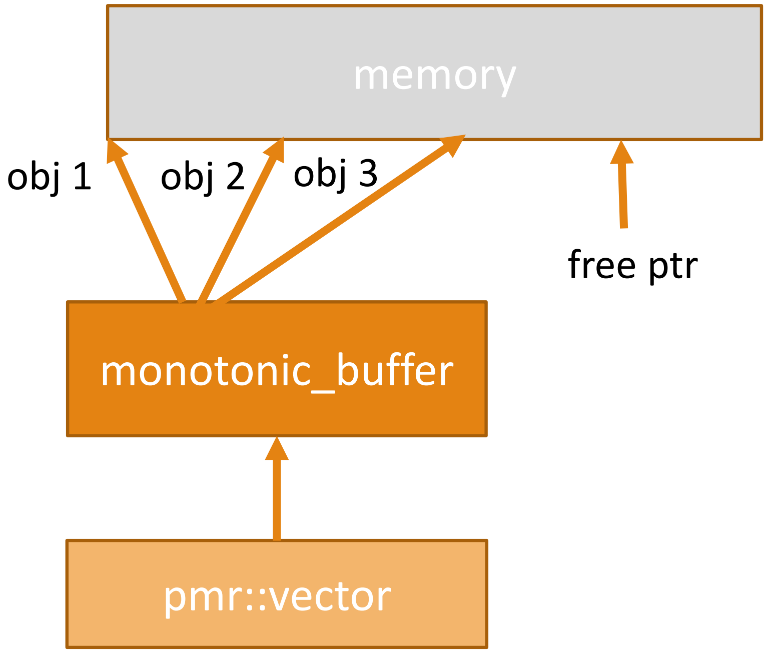
Monotonic buffer resource, contd.

- Basic usage

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

vec1.push_back("strg XXX");
vec2.push_back(44);
```

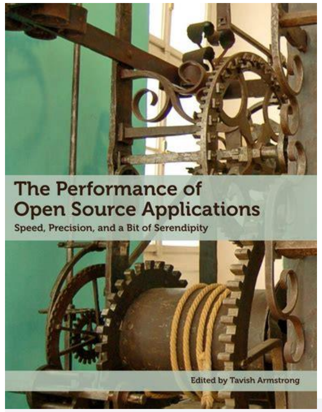
- **Caveat**: *std::pmr::monotonic_buffer_resource* in loops

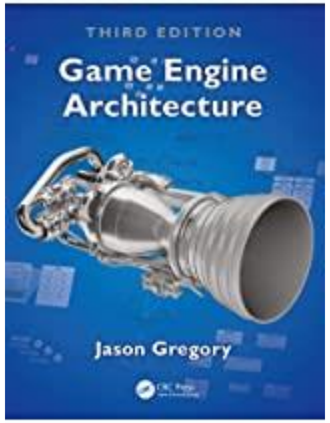
```
for (int i = 0; i < N; ++i)
{
   buffer_mem_res.release(); // rewind to the beginning!

   std::pmr::vector<std::pmr::string> strg_vec(&buffer_mem_res);

   strg_vec.push_back(std::format("strg {}-{}", i, i);
   // etc...

} // end of scope: strg_vec destroyed, but buffer_mem_res NOT shrinked!
```



Nihil novi sub sole

- XML parsing using lists for DOM representation [*Book 1*]
  - *Buffer-backed bump-up allocator* (but it can release elements)
  - List's performance problems solved!
- Game programming community: the classic [*Book 2*]
  - *stacked allocators* – memory cannot be freed in arbitrary order! (bump-up/down)
  - *single-frame allocators* – at the begin of each frame stacks top poiner is resetted!
- „Jak & Daxter" game
  - The loader 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 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





Basic PMRs 2: Pool resource

- *std::pmr::unsynchronized_pool_resource*
  - pool allocator - it consists of a collection of *pools* that serve requests for different block sizes.
  - fast allocations: finding the best-fit block easy!
  - eliminates fragmentation, maximizes locality
  - optimized for blocks of equal sizes
- *std::pmr::synchronized_pool_resource*
  - same pool allocator but with locks
- *std::pmr::pool_options*
  - can parametrize the pool resource

Constructors:

```
unsynchronized_pool_resource(); // uses default resource for upstream
unsynchronized_pool_resource(std::pmr::memory_resource* upstream);

unsynchronized_pool_resource(const pool_options& opts);
unsynchronized_pool_resource(const pool_options& opts, std::pmr::memory_resource* upstream);
```

Pool resource example implementation

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

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

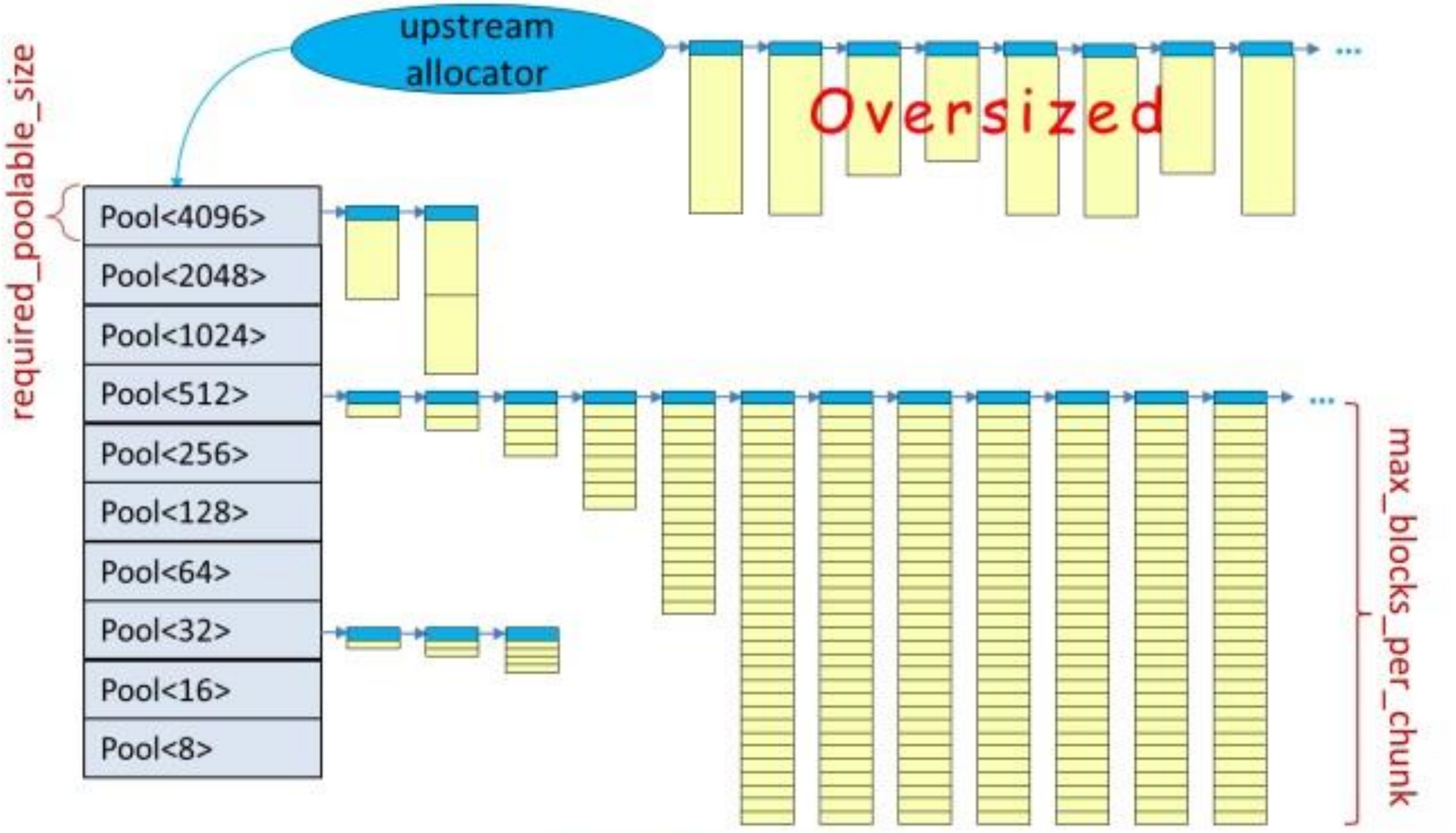
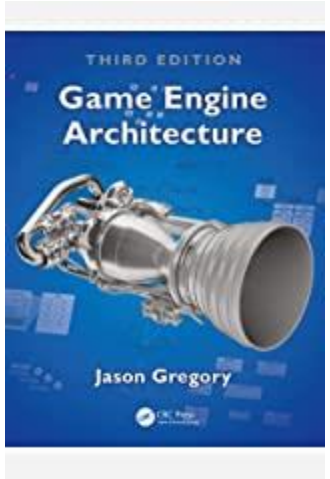


Figure 1: pool resource

– *end example*]

Nihil novi sub sole, contd.

- Game programming community: the classic [*Book*]
  - *pool allocators* - particles, projectiles, spaceships, etc.
    - They extremely simplify the allocation logic
    - Also they cater for **good memory locality**! You would like to have all of your particles in a nice array, right?
- 1st usage I‘ve seen - sized blocks allocators
  - e.g. for network packets: overloading C++ *new() operator* in a class
- Apache HTTP server
  - Memory pools: *„a collection of fixed sized elements called memory blocks… Unlike the heap, …at the mercy of other code …pools can insure sufficient memory allocation“**
  - *„Explicit regions were instrumental in the design some early C-based software projects, including the Apache HTTP Server, which calls them pools“*
  - Pool hierarchy for separation and locality!



APACHE HTTP SERVER

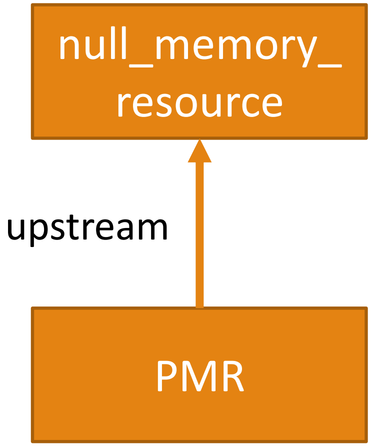
* *https://mynewt.apache.org/latest/os/core_os/memory_pool/memory_pool.html*

Special PMRs: Null memory resource

→ Is it of any use?

- It doesn‘t allocate, thus it will:
  - throw *std::bad_alloc* whan *allocate()* is called
- Thus we can use it as a kind of allocation guard:

```
// allocate memory on the stack
std::array<std::byte, 20000> buff;

// pool *without* fallback memory allocation on heap
std::pmr::monotonic_buffer_resource pool{ buff.data(), buff.size(),
                                          std::pmr::null_memory_resource() };
```



Choosing a PMR type for the task

Well-known memory tricks

→ There are several tricks we can use to speed things up:

- If we need a variable sized array/vector:
  - allocate array on the stack instead of heap (Linux: *alloca()*, Windows: _*alloca()*).
  - top of the stack is almost always in cache and super fast to allocate and deallocate!

  monotonic_buffer

- If we allocate many small memory blocks using *malloc()*
  - allocate one large block and then split it into smaller ones on your own
  - we shave off the overhead of repeatedly calling *malloc()*!

  monotonic_buffer

- If we allocate/deallocate many objects of the same type
  - cache memory blocks instead returning them with *free()*
  - they will be then easily reused if needed later!

  pool

Scenario 1: short-lived dynamic object

- **Use case:**
  A short-lived dynamic object, only used locally, e.g. a temporary string
- **Recommendation:**
  use *std::pmr::monotonic_buffer_resource* backed with a memory buffer on stack

```
void func()
{
  unsigned buffer[1024] = {};
  std::pmr::monotonic_buffer_resource buffRes(buffer, sizeof(buffer));

  std::pmr::string temp(&buffRes);
  temp = "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC";
  temp += "BBBBBBBBBB";

  // ...
 }
```

→ This might bypass the global heap entirely!

Scenario 2: large data structure, rarely changed

- **Use case:**
  Large data structure, built-up mononotically (elements added, but rarely, if ever, removed)
- **Recommendation**:
  use *std::pmr::monotonic_buffer_resource* as it is designed for that use case!
  - if possible call *reserve(expected_max_value)* on the container to avoid reallocations!

```
std::pmr::monotonic_buffer_resource buffRes;

std::pmr::unordered_map<std::pmr::string, int> configMap(&buffRes);
configMap.reserve(MAX_CONFIG_ENTRIES_COUNT):

readConfig(configMap);

int maxConns = getConfigVal(configMap, "maxConnectionCount");

// etc...
```

Scenario 3: data structure with frequent updates

- **Use case:**
  Data structure with frequent insertions and deletions, long-lived
- **Recommendation:**
  - use *std::pmr::unsynchronized_pool_resource* to enable efficient memory reuse with good locality
  - Chaining a *monotonic_buffer_resource* upstream sometimes yields noticeable performance gains over either one alone!

```
std::pmr::monotonic_buffer_resource buff;
std::pmr::unsynchronized_pool_resource pool(&buff); // set the upstream resource

std::pmr::map<std::pmr::string, unsigned> userMap(&pool);

addUser(userMap, "Mr. X", 0);
removeUser(userMap, "Dr. Who");
incrementUserTime(userMap, "Mr. X", 44);

// etc...
```

Scenario 4: local objects in deep call hierarchy

- **Use case:**
  We create numerous local objects within a deep call hierarchy (possibly even a recursive one)
- **Recommendation:**
  Create a top-level *std::pmr::unsynchronized_pool_resource* and pass it down the call chain!
- **Explanation:**
  on the return from a function, the blocks returned to the pool (still hot in cache!) are ready for immediate reuse in the next function call

```
std::pmr::monotonic_buffer_resource buff;
std::pmr::unsynchronized_pool_resource pool(&buff); // set the upstream resource

void func(std::pmr::memory_resource_resource* memRes)
{
   std::pmr::string s(memRes);

   recursiveFuncA(s, memRes);
   recursiveFuncB(s + “44" , memRes);

   // etc ...
}
```

Scenario 5 (**advanced**): Wink-out

- **Trick:** if we use a monotonic buffer for the container itself…
  - no destructors called for elements of the container!
  - because *deallocate()* method in *pmr::monotonic_buffer_resource* is a no-op
  - of course, *caution*: no side effects on destructors!!!

```
{
   std::pmr::monotonic_buffer_resource buff_resrc();
   std::pmr::polymorphic_allocator<> buff_alloc(&buff_resrc);

   auto& data = *buff_alloc.new_object<std::pmr::vector<std::pmr::list<std::pmr::string>>>();

   data.push_back({});
   data.push_back({});

   data[0].push_back("string XXX");
   data[0].push_back("string YYY");
   data[1].push_back("string ZZZ");

   // 'data' leaked? No, winked-out!
}
```

- Google *ProtoBuf*'s arenas:
  - ➢ *"Objects can all be freed at once by discarding the entire arena, **ideally without running destructors** of any contained object… "* :-o

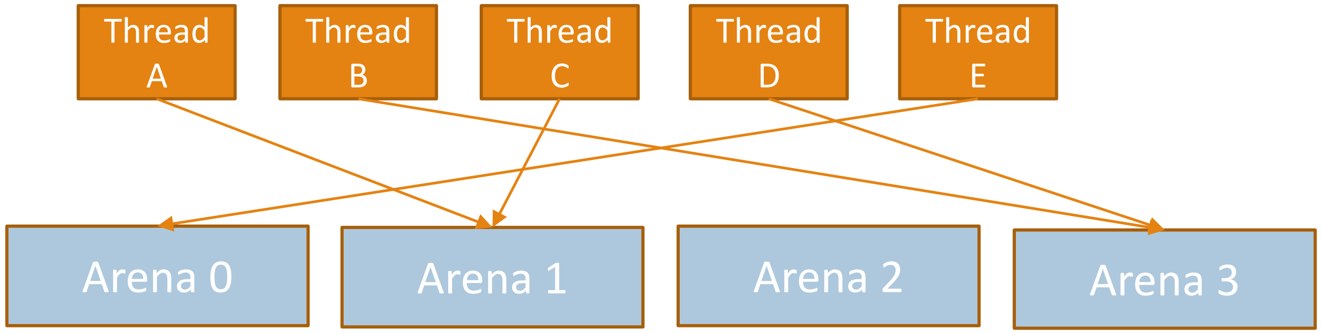
Tuning jemalloc

jemalloc

- *jemalloc* is a general purpose *malloc(3)* implementation intended for use as system allocator
  - In system's *libc* library, as well as for linking into C/C++ applications.
- It emphasizes avoidance of fragmentation and scalable concurrency support.
- It provides introspection, memory management, and tuning features
  - far beyond the standard allocator functionality!
- Major users of *jemalloc* include:
  - FreeBSD
  - Mozilla Firefox
    - "…*jemalloc* gave us the **smallest amount of fragmentation** after running for a long period of time. [...] Our automated tests on Windows Vista showed **a 22% drop in memory usage** when we turned *jemalloc* on."
  - Facebook (*folly::fbvector* connection!)
  - Databases: Cassandra, MariaDB
  - Android, etc.

jemalloc‘s design

- *jemalloc* uses multiple arenas (4 per CPU) in order to reduce lock contention for threaded programs
  - arenas manage memory completely independently of each other!



- It differentiates between three size categories: small, large and huge
  - These categories are further split ito diffrent size classes, which are kept in separare „runs“
    - E.g. Small: 8, 16, 32, 48, 64, 80, 96, 112, 128, 160, 192, 224, 256, etc.…
- Additionally, this allocator supports thread-specific caching for small and large objects
  - speeds-up allocation, but increases fragmentation
- It uses low address reuse to reduce fragmentation

What can we tune – runtime options

- **background_thread**
  Enables clock-bases purging of dirty pages in a dedicated background threads
- **dirty_decay_ms** and **muzzy_decay_ms**
  Decay time determines how fast *jemalloc* returns unused pages back to the operating system, and therefore provides a fairly straightforward trade-off between CPU and memory usage.
- **Narenas**
  High arena count may increase overall memory fragmentation, since arenas manage memory independently. When high degree of parallelism is not expected at the allocator level, lower number of arenas often improves memory usage.
- **percpu_arena**
  Enable dynamic thread to arena association based on running CPU. This has the potential to improve locality, e.g. when thread to CPU affinity is present.
- **Extensive** debug tracing and statistics!

Examples

- **High resource consumption** application, prioritizing **CPU utilization**:
  - *background_thread:true, metadata_thp:auto*
  - combined with longer decay time (e.g. *dirty_decay_ms:30000,muzzy_decay_ms:30000*).

- **High resource consumption** application, prioritizing **memory usage**:
  - *background_thread:true, tcache_max:4096.*
  - combined with shorter decay time (e.g. *dirty_decay_ms:5000,muzzy_decay_ms:5000*)
  - and lower arena count (e.g. number of CPUs)

- **Low resource** consumption application:
  - *narenas:1, tcache_max:1024*
  - combined with shorter decay time (e.g. *dirty_decay_ms:1000,muzzy_decay_ms:0*)

- **Extremely conservative** -- minimize memory usage at all costs
  - only suitable when allocation activity is very rare!
  - *narenas:1, tcache:false, dirty_decay_ms:0, muzzy_decay_ms:0*

What can we tune - programmatically

- Explicit usage of arenas/thread caches
  - We can manually create new arenas and thread caches and access them with *mallocx()*
  - application can allocate frequently accessed objects in a dedicated arena to improve locality!
  - explicit arenas can benefit from individually tuned options, e.g. relaxed decay time if frequent reuse is expected
- Explicit thread-to-arena binding
  - binding of very active threads to dedicated arenas may reduce contention at the allocator level!
- Extent hooks
  - These are hooks that allow customizations for managing underlying memory
  - Example use case: utilization of huge pages to reduce TLB misses
    - Facebook's HHVM uses explicit arenas with customized extent hooks to manage 1GB huge pages for frequently accessed data

Comparison with PMRs

→ *jemalloc*'s documentation says:

*"As an **extreme example**, arenas can be used as pool allocators; i.e. an arena can be used for general purpose allocation, and then the entire arena destroyed as a single operation."*

- Similar to Apache's memory pools!
  - Or our *synchronized_pool_resource* (because of locking overhead)
  - Nice possibility of pinning an arena to a CPU/thread
- Also thread caches can be used for unsynchronized memory allocations
  - However without the *monotonic_buffer_resource* mechanics
  - Size must be configured
- Thus more usable as an upstream allocator for our PMRs!
  - As in case of Facebook's *folly::fbvector*
  - But also as kind of „tagged heap" allocator

Summing Up

Conclusion: allocators

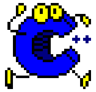
- Writing custom data structs/allocators – inherently costly
  - but every developer can just use *std::pmr* ones! ☺
  - predefined pool and monotonic PMRs cover most of the use cases ☺

- Customize/tune the system allocator instead?
  - OK, possible, but rather hard
  - However interesting as backing option for PMRs

→ **Always**: measure first, don‘t optimize blindly!

- Not only for improving performance, but also for:
  - placing objects on the stack / in file mapped memory
  - measuring / reporting memory usage
  - testing correctness (C++23?)

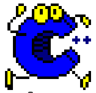
Allocator limitations and problems

→ Problems? Seriously? 

- Well, we don’t have:
  - *pmr::shared_memory_resource* - we must use *Boost.Interprocess* allocators! (Planned?) :-(
  - *pmr::test_resource* - not yet there (only proposal, C++23?)
- Although it has an allocator argument, *shared_ptr* 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!
- In C++14, *std::function* had several constructors taking an allocator argument
  - but they were removed (!) per P0302R1 because:

  "the semantics are unclear, and there are technical issues with storing an allocator in a type-erased context and then recovering that allocator later for any allocations needed during copy assignment" :-o
- In C++17, *std::any* does not allow allocator customization at all!

Future?

→ This all is veeery boring! 
→ Why not let the compiler do the work?

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

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

- HALO – Post-Link Heap Layout Optimization
  - Like LTO / PGO – profile guided
  - or annotation-based
  - research topic, long way to go… :-\

Thank you!

Any questions?

Bloomberg allocator‘s library design

- Before STL/C++98:
  - They had developed a set of stateful, generally non-equal custom allocators (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 > > >* !!!

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

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*

PMRs and smart pointers

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

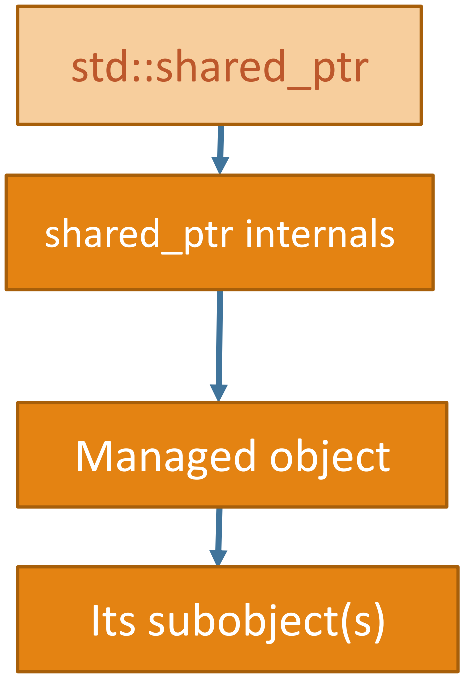
```
template< class Y, class Deleter, class Alloc >
    shared_ptr( Y* ptr, Deleter d, Alloc alloc );
```

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!



Erased Allocators (C++ 11)

- *std::function, std::promise, std::future …* have got it in C++11
  - NOT chosen for memory resources!
- type erasure in C++ howto:
  - don‘t use *T* in class type, use a template constructor and a wrapper instead!

```
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 )) { }
```

Thank you again!

Any questions?