Getting Allocators out of Our Way

Alisdair Meredith and Pablo Halpern

CppCon 2019

Allocators are great (as you know)!

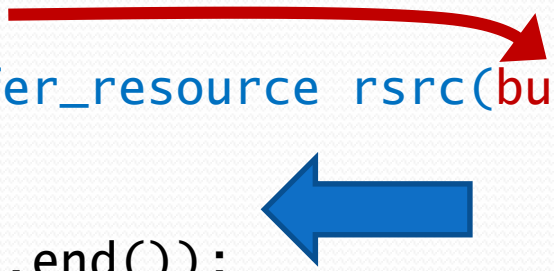
- Without allocators

```
std::size_t unique_chars(std::string_view s)
{


  std::set<char> uniq;
  uniq.insert(s.begin(), s.end());
  return uniq.size();
}
```

Allocators are great (as you know)!

- With C++17 PMR allocators

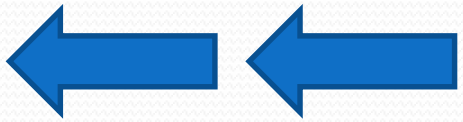
```
std::size_t unique_chars(std::string_view s)
{
  std::byte buffer[4096];
  std::pmr::monotonic_buffer_resource rsrc(buffer, sizeof buffer);

  std::set<char> uniq;
  uniq.insert(s.begin(), s.end());
  return uniq.size();
}
```



Allocators are great (as you know)!

- With C++17 PMR allocators

```
std::size_t unique_chars(std::string_view s)
{
  std::byte buffer[4096];
  std::pmr::monotonic_buffer_resource rsrc(buffer, sizeof buffer);

  std::pmr::set<char> uniq();
  uniq.insert(s.begin(), s.end());
  return uniq.size();
}
```



Allocators are great (as you know)!

- With C++17 PMR allocators

```
std::size_t unique_chars(std::string_view s)
{
  std::byte buffer[4096];
  std::pmr::monotonic_buffer_resource rsrc(buffer, sizeof buffer);

  std::pmr::set<char> uniq(&rsrc);
  uniq.insert(s.begin(), s.end());
  return uniq.size();
}
```

- Observed: a 5x speedup!

But there are costs

- Costs *are not* in using allocators
- Costs *are* in infrastructure & middleware
  - Developing, plumbing *allocator-aware (AA)* classes
  - Getting it wrong

(CC0) Pixels

But there are costs

- Costs *are not* in using allocators
- Costs *are* in infrastructure & middleware
  - Developing, plumbing *allocator-aware (AA)* classes
  - Getting it wrong
- Today's *AA* software isn't perfect
  - Incompatible with some language features

(CC0) Pixels

What if we could reduce the cost?

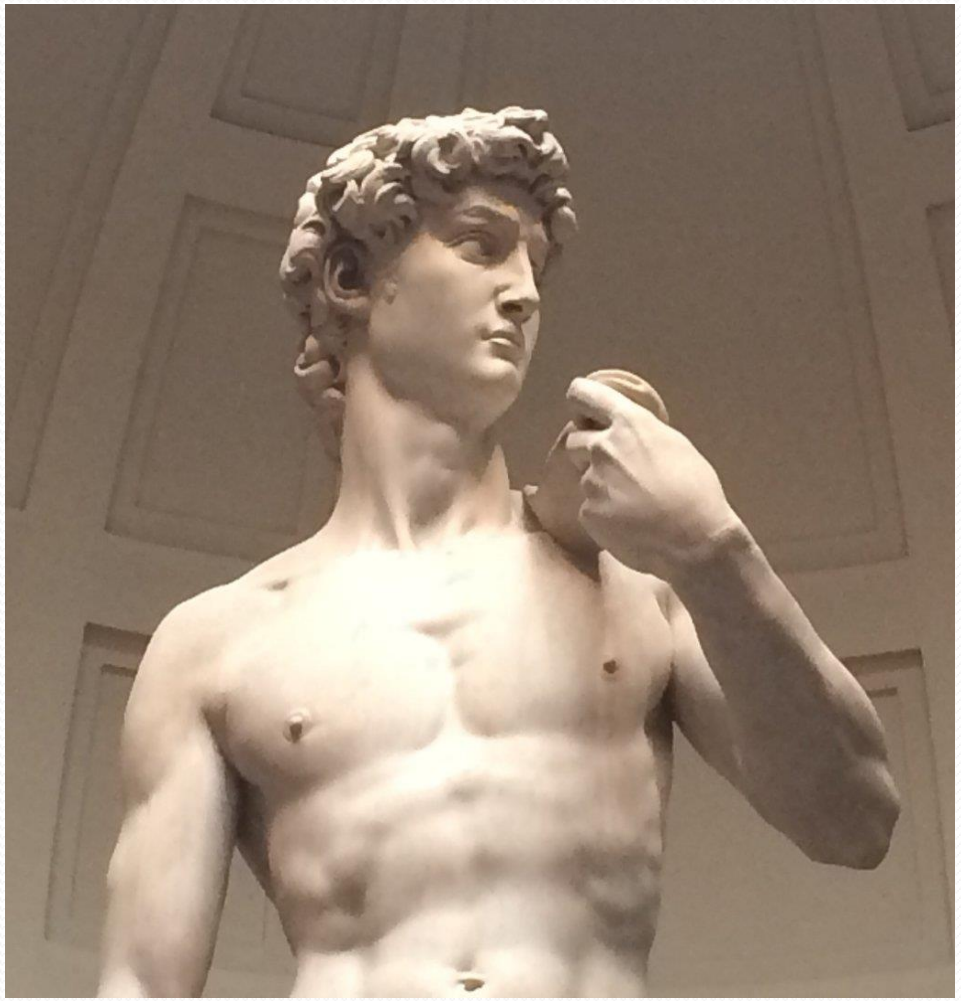
- Sneak preview of our language proposal

- Goal: To reduce the cost of developing an allocator-aware class almost to zero

Image: Public Domain
(Courtesy Wikipedia)

What if we could reduce the cost?

- Sneak preview of our language proposal
- Goal: To reduce the cost of developing an allocator-aware class almost to zero
- While we're at it, make *AA* software almost perfect



Who are we?

Pablo Halpern

- Independent software consultant
- C++ Standards Committee Member
  - To blame for much of the allocator stuff
- Sixth year presenting at CppCon
- This is my nerdy car

Alisdair Meredith

- Senior engineer at Bloomberg LP
- C++ Standards Committee Member
  - Was complicit in the allocator stuff
- Fifth year presenting at CppCon
- This was my last car

What's our problem?

Memory is the *oxygen* of computing!

The air we breathe is invisible

*(yet the allocators we use **today** are anything but invisible)*

Allocators bloat class interfaces

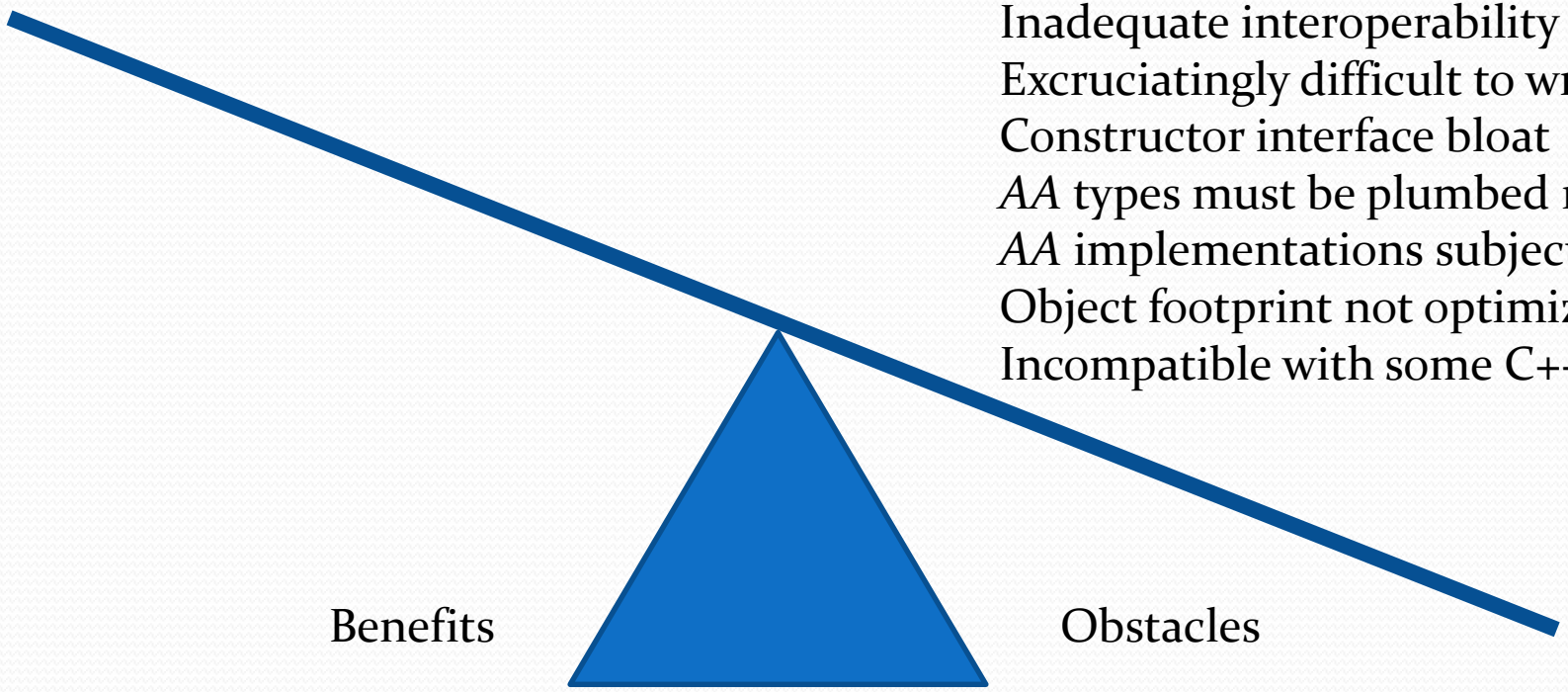
(yet, for many, allocators are pure implementation details)

Allocators pervade class implementations

(yet typically only as "mindless" boilerplate)

Allocators are in our way!

C++98/03 allocators

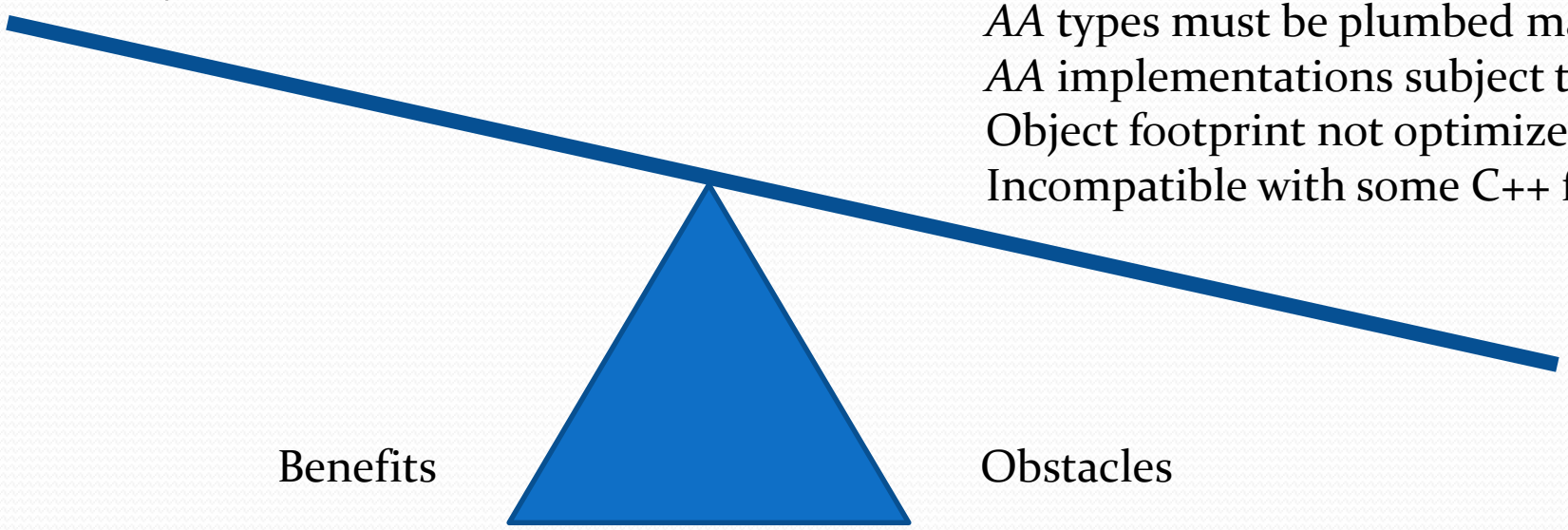


Incomplete specification
Non-portable support for stateful allocators
Inadequate interoperability at scale
Excruciatingly difficult to write
Constructor interface bloat
AA types must be plumbed manually
AA implementations subject to human error
Object footprint not optimized by compiler
Incompatible with some C++ features

C++11 allocators

Runtime performance
Scoped allocator model
Localized (“arena”) object memory
Entire-object placement in memory
Per-object metrics/measurement

Inadequate interoperability at scale
Difficult to write
Constructor interface bloat
AA types must be plumbed manually
AA implementations subject to human error
Object footprint not optimized by compiler
Incompatible with some C++ features



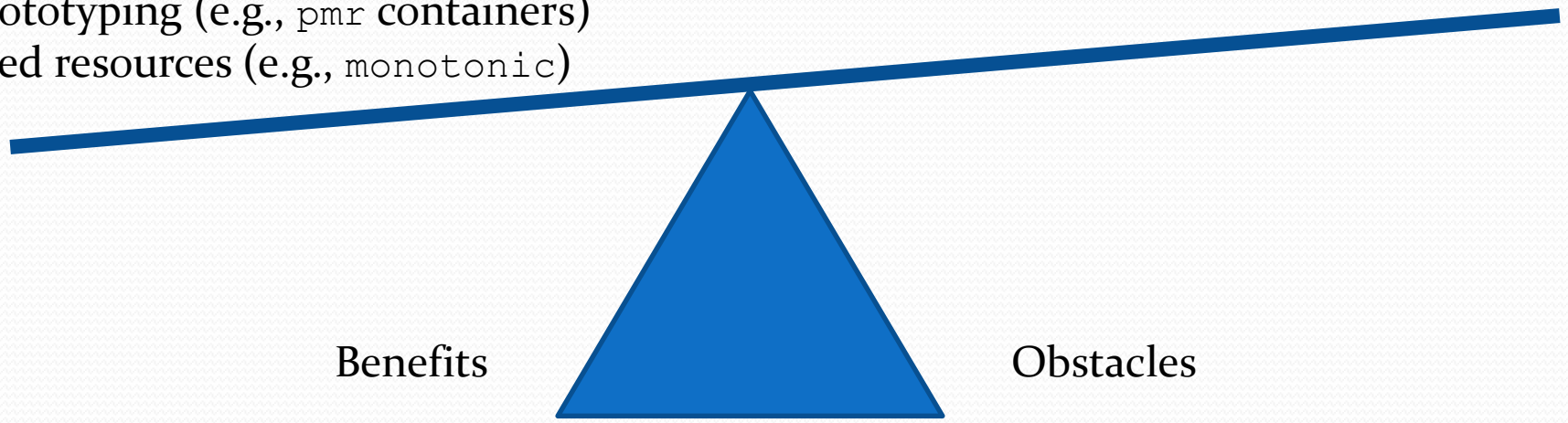
Benefits

Obstacles

C++17/20 allocators

Runtime performance
Scoped allocator model
Localized ("arena") object memory
Entire-object placement in memory
Per-object metrics/measurement
Ubiquitous vocabulary types (handles)
Simple (to write/use) allocators
Rapid prototyping (e.g., `pmr` containers)
Predefined resources (e.g., `monotonic`)

Constructor interface bloat
AA types must be plumbed manually
AA implementations subject to human error
Object footprint not optimized by compiler
Incompatible with some C++ features

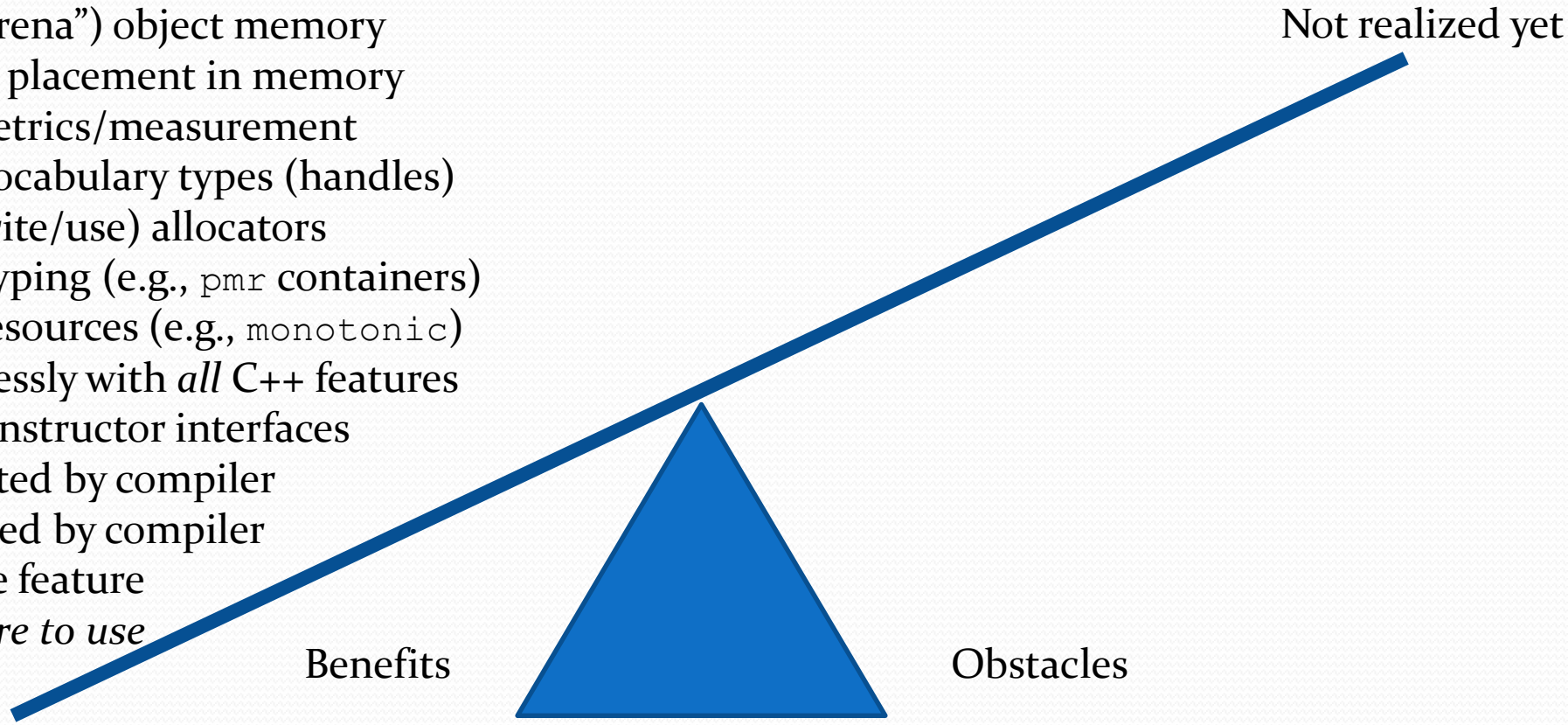


Benefits

Obstacles

Our Goal (Not Yet Realized)

Runtime performance
Scoped allocator model
Localized ("arena") object memory
Entire-object placement in memory
Per-object metrics/measurement
Ubiquitous vocabulary types (handles)
Simple (to write/use) allocators
Rapid prototyping (e.g., `pmr` containers)
Predefined resources (e.g., `monotonic`)
Works seamlessly with *all* C++ features
Simplified constructor interfaces
Fully automated by compiler
Fully optimized by compiler
Generalizable feature
A true pleasure to use



Agenda

- Why allocators?
- How have they improved?
- How are they still in our way?
- What language changes might get them out of our way?

At the end of this talk

- We want you to understand the good and the bad of allocators today
- We want you to be excited about our proposal
- We want your feedback
- We want your undying devotion to our cause!

Let's get started

Performance in practice

- Memory is arranged in a hierarchy
  - Cache (L1, L2, L3) ⇐ main memory ⇐ virtual memory on disk
  - Each layer is 1 to 3 orders of magnitude faster than the next
- Objects that are used together should reside close to each other
- Diffusion can reduce the performance of long running programs

Why do we need custom memory allocation?

- Performance
- To place objects in special memory
- To instrument memory allocation
  - Gathering statistics
  - Testing
- But mostly, for performance

With custom memory allocation, we could

- allocate related objects in contiguous pages
- avoid concurrency locks within the same thread
- reduce fragmentation and diffusion
- place objects in special memory locations, e.g.,
  - persistent memory
  - high-bandwidth memory
- gather data on allocations and deallocations

Warning C++11 allocators ahead

C++11 Allocators

- Encapsulate a custom allocation strategy

```
template <typename Tp>
class MyAlloc {
  [[nodiscard]] Tp* allocate(std::size_t n);
  void deallocate(Tp* p, std::size_t n);
  …
};
```

- Can be plugged into any of the standard containers

```
MyAlloc<int> alloc(buffer, buffsize);
std::vector<int, MyAlloc<int>> v(alloc);
```

- Since C++11, we have had full support for stateful allocators

But allocators are complicated!

- Typedefs
  - `value_type`
  - `pointer`
  - `const_pointer`
  - `difference_type`
  - `size_type`
- Propagation traits
  - `select_on_container_copy_construction`
  - `propagate_on_container_copy_assignment`
  - `propagate_on_container_move_assignment`
  - `propagate_on_container_swap`
  - `is_always_equal`
- Nested template
  - `template <class T> rebind`
- Member functions
  - `allocate`
  - `deallocate`
  - `construct`
  - `destroy`
  - `max_size`
- Non-member operators
  - `operator==`
  - `operator!=`

allocator_traits simplify allocators

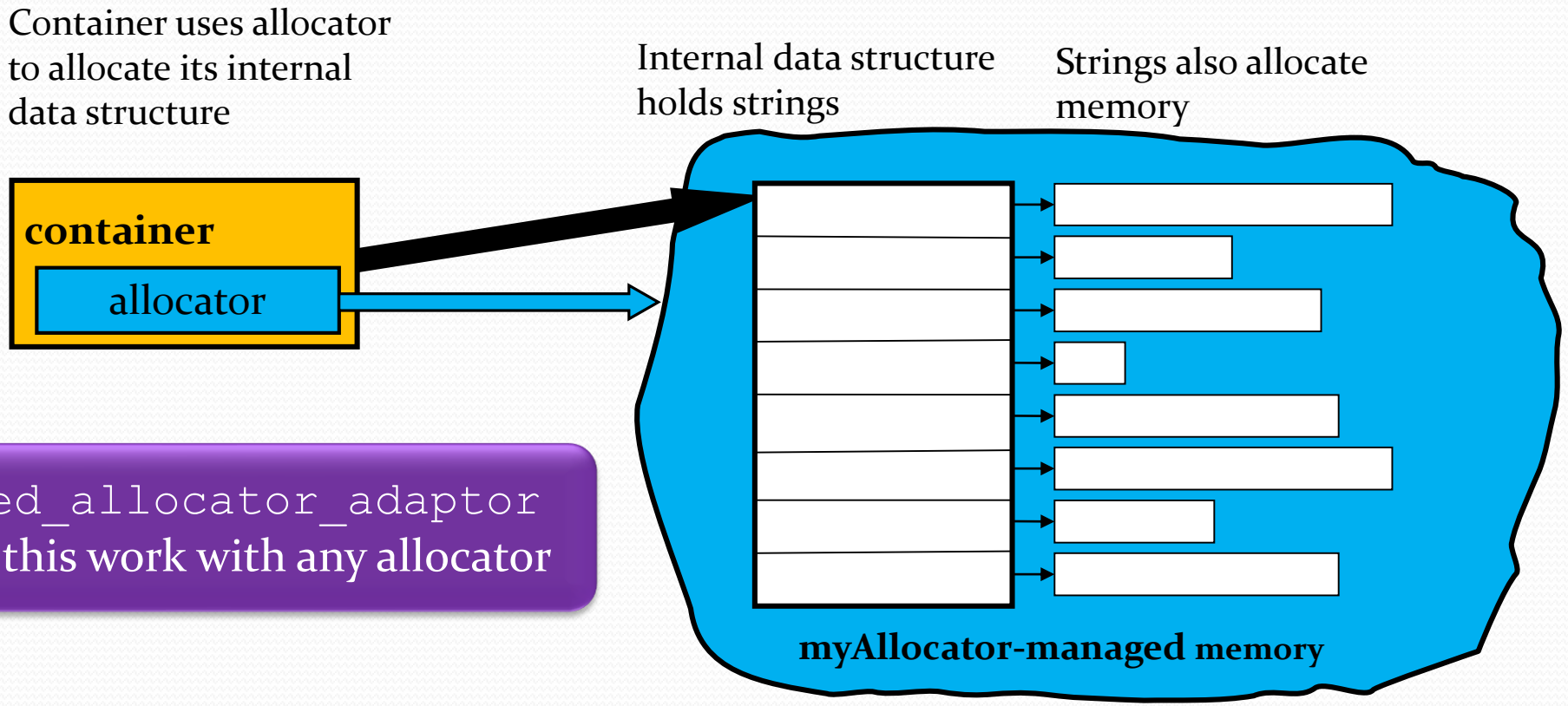
- Typedefs
  - value_type
  - pointer
  - const_pointer
  - difference_type
  - size_type
- Propagation traits
  - select_on_container_copy_construction
  - propagate_on_container_copy_assignment
  - propagate_on_container_move_assignment
  - propagate_on_container_swap
  - is_always_equal
- Nested template
  - template <class T> rebind
- Member functions
  - allocate
  - deallocate
  - construct
  - destroy
  - max_size
- Non-member operators
  - operator==
  - operator!=

What about nested containers?

- In a `vector<string>`
  - The `vector` has one allocator
  - Each `string` could have a different allocator
- No physical coherency –no real control over allocation for the whole container
- Hard to track lifetimes of all of those allocators

The Scoped Allocator Model

```
vector<string> container(myAllocator)
```



`scoped_allocator_adaptor` makes this work with any allocator

The Scoped Allocator Model


```
vector<string> container(myAllocator)
```

Container uses allocator

Internal data structure ... Strings also allocate

Actually:

```
vector<basic_string<char, char_traits<char>,
                    scoped_allocator_adaptor<MyAlloc<char>>>,
       scoped_allocator_adaptor<
         MyAlloc<basic_string<char, char_traits<char>,
                              scoped_allocator_adaptor<MyAlloc<char>>>>
         >
       > container(myAllocator)
```

The allocator is part of the type

- This won't work:

```
std::vector<int, std::allocator<int>>
```

```
void func(const std::vector<int>&);            // Default allocator
std::vector<int, MyAlloc<int>> v(someAlloc);   // Custom allocator
func(v);  // ERROR: v is a different type than std::vector<int>
```

- This works, but doesn't scale:

```
template <class Alloc> void func(const std::vector<int, Alloc>&);
```

- **Allocator template policies interfere with vocabulary types.**

C++17 PMR: a simpler allocator model

pmr = "polymorphic memory resource"

- Non-template `std::pmr::memory_resource` has `allocate` and `deallocate` member functions
- Thin wrapper `std::pmr::polymorphic_allocator<T>` meets the C++11 allocator requirements
- Each standard container has a `pmr` alias:

```
namespace std::pmr {
  template <class T>
    using vector = std::vector<T, polymorphic_allocator<T>>;
}
```

Polymorphic memory resources

```
namespace std::pmr {

class memory_resource {
public:
  virtual ~memory_resource();
  void* allocate(size_t bytes, size_t alignment = max_align);
  void deallocate(void* p, size_t bytes,
                  size_t alignment = max_align);
  bool is_equal(const memory_resource& other) const noexcept;
  …
};

}
```

delegate to virtual functions

Polymorphic allocator

```
namespace std::pmr {

template <class Tp>
class polymorphic_allocator {
  memory_resource *rsrc;

public:
  polymorphic_allocator() noexcept : rsrc(get_default_resource()) {}
  polymorphic_allocator(memory_resource* r) : rsrc(r) {}
  memory_resource* resource() const { return rsrc; }

  Tp* allocate(size_t n) { return (Tp*) rsrc->allocate(n * sizeof(Tp)); }
  void deallocate(Tp* p, size_t n) { rsrc->deallocate(p, n * sizeof(Tp)); }
  …
};

}
```

An allocator vocabulary type

- The resource type is determined ***at run time***
  - The compile-time type of all `pmr::vector<int>` are the same
  - Interoperability problems disappear

C++11: Doesn't work	C++17 PMR: Works!

`void func(const std::vector<int>&);`
`std::vector<int, MyAlloc<int>> v(anAlloc);`
`func(v);`
	`void func(const std::pmr::vector<int>&);`
	`std::pmr::vector<int> v(anAlloc);`
	`func(v);`

polymorphic_allocator<byte>

- The “one true” allocator vocabulary type
- In C++20, abbreviated as `polymorphic_allocator<>`
- No viral template explosion
- scoped allocator model
  - Allocator passes itself to sub-objects

Where do I get `memory_resource` classes?

C++17 provides standard memory resources!

std::pmr::new_delete_resource()

- Returns a resource that delegates to the general heap (using `operator new` and `operator delete`)
- Always available, general purpose, and thread safe
- Used by default if no other resource is specified

std::pmr::unsynchronized_pool_resource

- Pools of similar-sized objects ensure excellent spatial locality
- Good for dynamic data structures that grow and shrink
- Single-threaded*
  - Appropriate only for non-concurrent containers (most containers)
  - Cannot be used simultaneously by containers in different threads
  - Avoids concurrency lock overhead

* multithreaded `std::pmr::synchronized_pool_resource` also exists, but is not very efficient

std::pmr::monotonic_buffer_resource

- Ultra-fast, single-threaded, allocate from contiguous buffers and do-nothing deallocate.
- For containers that grow monotonically throughout their lifetime
- Can allocate from stack memory

```
void f()
{
  std::byte stackBuf[2048];
  std::pmr::monotonic_buffer_resource rsrc(stackBuf, sizeof stackBuf);

  std::pmr::list<Thing> listOfThings(&rsrc);

  insertThings(&listOfThings);
  processThings(&listOfThings);
} // All memory is released here
```

Test resource

- Provides useful features for testing & debugging
  - Catch allocator misuse
  - Collect usage statistics
  - Validate exception-resiliency

- Not standard yet, but proposed for Library Fundamentals 3:
  - Open source implementation: https://github.com/bloomberg/p1160
  - See Attila Fehér's talk “test_resource: The pmr Detective”, tomorrow (Thursday) at 16:45

Composing resources into a chain

- Each standard resource can be constructed with an *upstream resource* for replenishing its internal pools

```
std::pmr::monotonic_buffer_resource mrsrc(buffer, size);
std::pmr::unsynchronized_pool_resource prsrc(&mrsrc);
test_resource trsrc(&prsrc);
```

Now we have Allocators!
What about classes that use them?

Simple classes that use allocators

- A simple class with data members that allocate:

```
class Student {
  string             d_name;
  string             d_email;
  vector<int>        d_grades;

public:

  Student(string_view name, string_view email)
    : d_name(name)
    , d_email(email)
    , d_grades() { }
  …
};
```

Adding allocator constructor arguments

- Add an optional (defaulted) allocator argument to each constructor:

```
class Student {
  pmr::string       d_name;
  pmr::string       d_email;
  pmr::vector<int> d_grades;

public:
  using allocator_type = pmr::polymorphic_allocator<>;
  Student(string_view name, string_view email, allocator_type alloc = {})
    : d_name(name, alloc)
    , d_email(email, alloc)
    , d_grades(alloc) { }
  …
};
```

Interaction with Default Arguments

- Overload each constructor having default arguments:

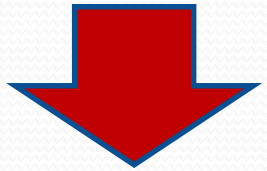
```
explicit Student(string_view name, string_view email = "no-email",
                 const pmr::vector<int>& grades = {});
```

```
explicit Student(string_view name, allocator_type alloc = {});
Student(string_view name, string_view email, allocator_type alloc = {});
Student(string_view name, string_view email,
        const pmr::vector<int>& grades, allocator_type alloc = {});
```

Interaction with variadic argument lists

- The allocator goes first for constructors with variadic argument lists:

```
template <typename... Args>
  explicit Student(string_view name, Args&&... grades);
```



```
template <typename... Args>
  explicit Student(string_view name, Args&&... grades);
template <typename... Args>
  explicit Student(allocator_arg_t, allocator_type alloc,
                   string_view name, Args&&... grades);
```

A larger example: constructor overload!

- Let's write `std::unordered_map`, simplified to use `pmr` allocators
  - Change the name to protect the innocent

No allocator template policy

```
template <class Key, class Value,
          class Hash = std::hash<Key>, class Equal = std::equal_to<Key>>
class HashMap {

public:
  using allocator_type = std::pmr::polymorphic_allocator<>;
  using Alloc = allocator_type;    // shorter name for the next few slides!
  // Public typedefs
  …

  // Constructors
```

HashMap constructors

```
HashMap();
explicit HashMap(Alloc);

HashMap(const HashMap&);
HashMap(const HashMap&, Alloc);

HashMap(HashMap&&);
HashMap(HashMap&&, Alloc);

explicit HashMap(size_t n, Hash hf = {}, Equal eq = {}, Alloc a = {});

template<class InputIterator>
HashMap(InputIterator first, InputIterator last, size_t n = 13,
        Hash hf = {}, Equal eq = {}, Alloc a = {});

HashMap(std::initializer_list<value_type> il, size_t n = 13,
        Hash hf = {}, Equal eq = {}, Alloc a = {});
```

Even more HashMap constructors

```
HashMap(size_t n, Alloc a) : HashMap(n, Hash{}, Equal{}, a) { }
HashMap(size_t n, const Hash& hf, Alloc a) : HashMap(n, hf, Equal{}, a) { }

template<class InputIterator>
HashMap(InputIterator first, InputIterator last, size_t n, Alloc a)
  : HashMap(first, last, n, Hash{}, Equal{}, a) { }

template<class InputIterator>
HashMap(InputIterator first, InputIterator last, size_t n, Hash hf, Alloc a)
  : HashMap(first, last, n, hf, Equal{}, a) { }

HashMap(std::initializer_list<value_type> il, size_t n, Alloc a)
  : HashMap(il, n, Hash{}, Equal{}, a) { }

HashMap(std::initializer_list<value_type> il, size_t n, Hash hf, Alloc a)
  : HashMap(il, n, hf, Equal{}, a) { }
…
```

We've come a long way, but…

- Improved cost/benefit for users
  - (C++11) Simplified allocator design (`allocator_traits`)
  - (C++11) Consistency for nested containers (scoped allocator model)
  - (C++17) Non-template vocabulary type (`polymorphic_allocator<byte>`)
  - (C++17) Pre-defined memory resources (e.g., `monotonic_buffer_resource`)

- Still significant costs for class developers
  - Allocators invade the interface and implementation (constructor overload)

- **Allocators are still in our way!**

We've reached the limits of a library approach

What about a language solution?

Towards a language proposal

- Build on our strong foundation
  - Non-template vocabulary (`memory_resource`)
  - Consistency for nested containers (scoped allocator model)
  - Pre-defined memory resources (e.g., `monotonic_buffer_resource`)

- Need to remove the obstacles to class implementation
  - Simplify the interface
  - Reduce implementation boilerplate

What if it were this simple?

```
// Constructors

HashMap(std::initializer_list<value_type> il = {},
        size_t n = 13, Hash hf = {}, Equal eq = {});

HashMap(const HashMap&);
HashMap(HashMap&&);

explicit HashMap(size_t n, Hash hf = {}, Equal eq = {});

template<class InputIterator>
HashMap(InputIterator first, InputIterator last,
        size_t n = 13, Hash hf = {}, Equal eq = {});
…
```

Out-of-band allocator argument

- What if an *allocator-aware* class could be constructed `using` an allocator
  - As if each constructor had a hidden allocator argument

```
HashMap<int, int> x{} using myAllocator;
HashMap<int, int> y{{1,2}, {3,4}, {5,6}, {7,8}} using myAllocator;
HashMap<int, int> z(y.begin(), y.end(), 41) using myAllocator;

HashMap<int, int> a using myAllocator = x;
HashMap<int, int> b using myAllocator = {{1,2}, {3,4}, {5,6}, {7,8}};

HashMap<int, int> c = HashMap<int, int>{} using myAllocator;
```

Properties of an *Allocator Aware* (*AA*) type

- Initialize with `using` *`allocator`*
  - Use the *default memory resource* when no allocator supplied
- Uses scoped allocator model

```
pmr2::HashMap<pmr2::string, pmr2::string> dictionary using anAllocator;
```

- Advertises its allocator

```
auto a = std::pmr2::get_allocator(dictionary);
```

- Trait for allocator awareness

```
static_assert(std::is_allocator_aware_v<std::pmr2::vector<int>>);
```

When is a class *AA*?

- A class should be *AA* if
  - It has any *AA* base classes
  - Or adds any *AA* data-members

- Compare: a class is *polymorphic* if
  - It has any polymorphic base classes
  - Or adds any virtual functions

It's a Kind of Magic

```
namespace std::pmr2 {

template <class T = byte>
class polymorphic_allocator;

}
```

- A library-supplied *AA* class
- `pmr2` is distinct from `pmr` to preserve compatibility

```
template <class Key, class Value,
          class Hash = std::hash<Key>, class Equal = std::equal_to<Key>>
struct HashMap {
  // constructors
  HashMap(std::initializer_list<value_type> il = {},
          size_t n = 13, Hash hf = {}, Equal eq = {});

  HashMap(const HashMap&);
  HashMap(HashMap&&);

  explicit HashMap(size_t n, Hash hf = {}, Equal eq = {});

  template<class InputIterator>
  HashMap(InputIterator first, InputIterator last,
          size_t n = 13, Hash hf = {}, Equal eq = {});
  // ...

private:
  std::pmr2::polymorphic_allocator<> d_allocator;
};
```

Fewer and simpler constructors

- Total number of constructors:

C++20 standard	15
Allocator aware	5

- Fewer overloads for explicitness
- Simple use of default arguments
- Plain copy and move constructors
- One fewer argument per constructor

- Bonus! Implicit code generation is one less opportunity to introduce bugs

Fixing a well-known idiom

```
class Object {
  std::pmr::string d_name;

public:
  using allocator_type = std::pmr::polymorphic_allocator<>;

  explicit Object(allocator_type a = {}) : d_name("<UNKNOWN>", a) {}
  Object(const Object& rhs, allocator_type a = {}) : d_name(rhs.d_name, a) {}
  Object(Object&&) = default;
  Object(Object&& rhs, allocator_type a) : d_name(std::move(rhs.d_name), a) {}

  // Apply rule of 6
  ~Object() = default;
  Object& operator=(const Object& rhs) = default;
  Object& operator=(Object&& rhs) = default;
};
```

Fixing a well-known idiom

```
class Object {
  std::pmr2::string d_name;

public:
  // using allocator_type = std::pmr::polymorphic_allocator<>;

  Object() : d_name("<UNKNOWN>") {}
  Object(const Object& rhs) = default;
  Object(Object&&) = default;
  // Object(Object&& rhs, allocator_type a)

  // Apply rule of 6
  ~Object() = default;
  Object& operator=(const Object& rhs) = default;
  Object& operator=(Object&& rhs) = default;
};
```

Fixing a well-known idiom

```
class Object {
  std::pmr2::string d_name = "<UNKNOWN>";

public:


  Object() = default;
  Object(const Object& rhs) = default;
  Object(Object&&) = default;


  // Apply rule of 6
  ~Object() = default;
  Object& operator=(const Object& rhs) = default;
  Object& operator=(Object&& rhs) = default;
};
```

Default member initializer

Fixing a well-known idiom

```
class Object {
  std::pmr2::string d_name = "<UNKNOWN>";

public:

  // Rule of zero




};
```

What if there are no constructors?

- Aggregates are special
  - Have no constructors*
  - Are initialized differently than non-aggregates

```
struct MyAggregate {
  int         x;
  std::string y;
  float       z;
};

MyAggregate a = { 1, "two", 3.14f };
MyAggregate b = { .y = "two", .z = 3.14f };    // C++20
```

AA aggregates

```
struct Student {
  std::pmr2::string       d_name;
  std::pmr2::string       d_email;
  std::pmr2::vector<int> d_grades;
};
```

- `using` will pass allocator to each initializer in turn

```
Student jLakos = {"John Lakos", "jlakos@uni.edu", {1, 2, 3}} using alloc;
```

- Aggregates do *not* need to store their own copy of allocator
  - `get_allocator` delegates to any allocator-aware element
- Arrays are also aggregates

```
std::pmr2::string x[] = {"one", "two", "three"} using alloc;
```

Language Integration

Language feature	C++20 Library-based allocators	Language proposal
Aggregates	Not supported	Supported
Arrays	Not supported	Supported
`std::array`	Not supported	Supported
Lambda expressions	Not supported	Supported
Structured bindings	Not supported	Supported
Inherited constructors	Limited	Supported
Coroutines	Not supported	Investigating…
`union`	Manually supported	Investigating…
Default member initializers	Ignored	Supported
Compiler-generated constructors	Will not use allocators	Supported

New ABI only for new types

Old types	No change	No Breakage
New *AA* types	New rules, new ABI	No Breakage
Old templates instantiated with *AA* types	New rules, new ABI	No Breakage expected

- Limited ability to change existing libraries
  - We will need one new set of vocabulary types
  - `std::pmr2::vector`, `std::pmr2::string`, etc.

Tally sheet

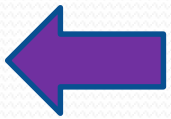
What have we removed?

- Interface bloat
- Implementation complexity
- Opportunities for programmer error

What have we gained

- Automation
- Language compatibility
- Ubiquitous availability of *AA* types
- Simplified interfaces
- Reliability

- We have ***not*** given up any performance or space efficiency!



Useful for things beyond allocators?

- Focusing on allocators for now
  - Avoid a premature generalization
  - Propose the best possible allocator feature

- Expect the final proposal to evolve for a wider variety of use cases
  - **executors**
  - stop tokens
  - network sockets
  - database connections
  - etc.

Not quite a roadmap

- Working on a C++23 proposal
  - cover all concerns before taking committee time
  - probably around 12 months away from submitting
  - mid-cycle entering C++23 EWG queue, so maybe C++26

- Working on a prototype compiler as we submit the proposal
  - Part of Bloomberg's 2020 Vision (BB2020V)
  - Flush out hidden assumptions
  - Measure real costs
  - Demonstrate actual benefits

Before we summarize…

Questions?

In Summary

- Allocators bring huge benefits to C++
  - Performance
  - Reliability
  - Instrumentation
- Allocators are easy to use…
  - … but there are real costs to implementing allocator-aware types

In Summary

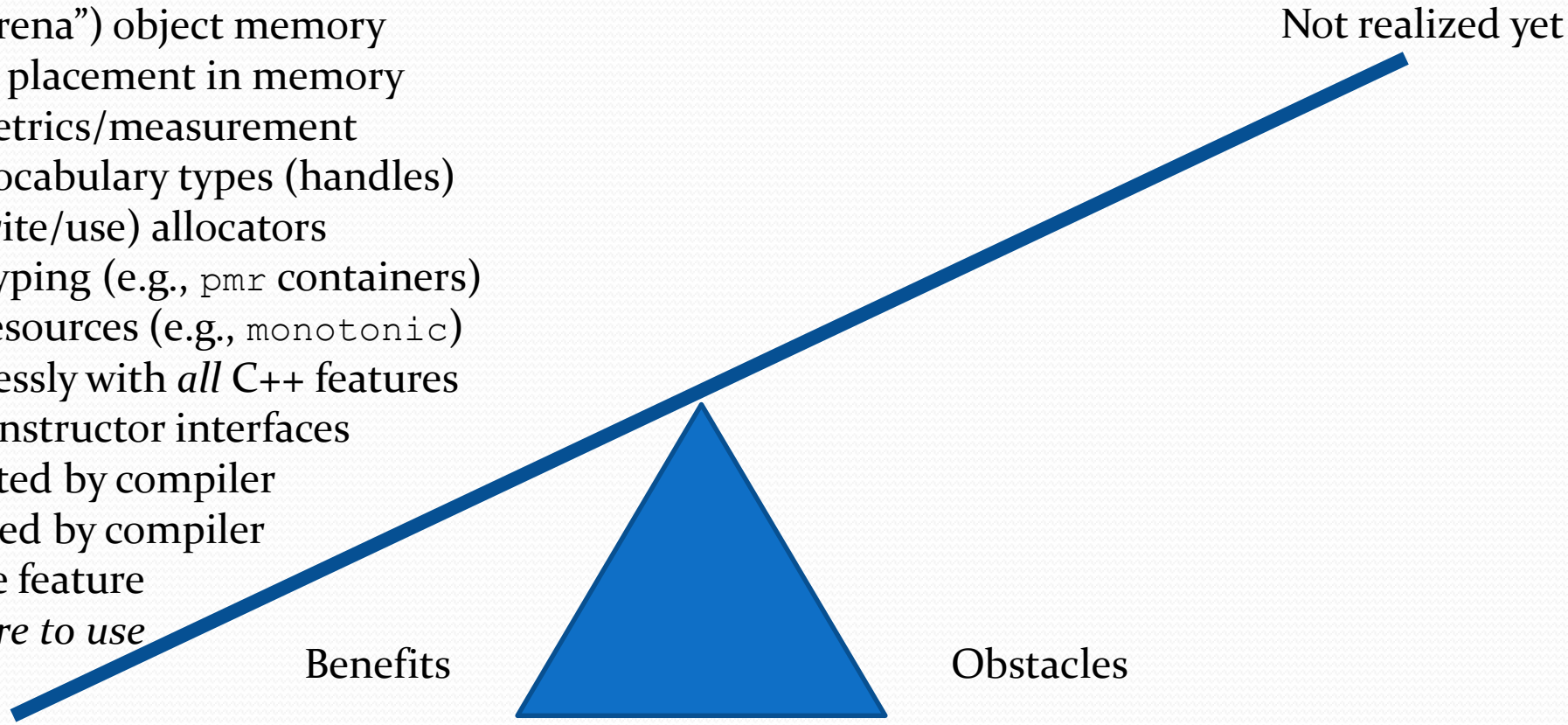
- Direct language support can reduce costs, nearly to zero
  - Less code = fewer bugs & simpler maintanance
  - Better integration with the rest of C++
  - Expected to increase use of allocators

In Summary

- Proposal and prototype compiler is in the works
  - Soliciting feedback
  - Looking for champions

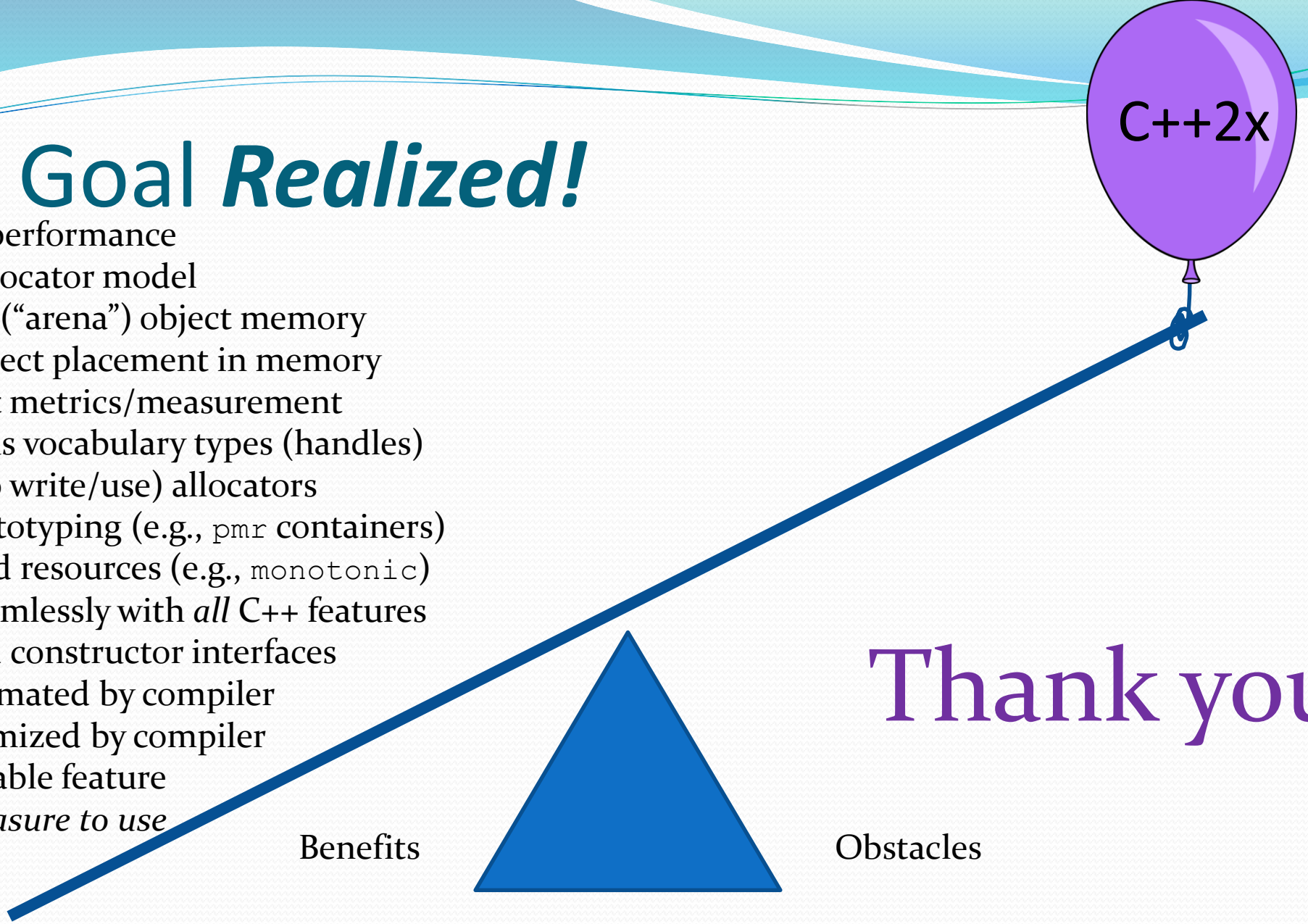
Our Goal (Not Yet Realized)

Runtime performance
Scoped allocator model
Localized ("arena") object memory
Entire-object placement in memory
Per-object metrics/measurement
Ubiquitous vocabulary types (handles)
Simple (to write/use) allocators
Rapid prototyping (e.g., `pmr` containers)
Predefined resources (e.g., `monotonic`)
Works seamlessly with *all* C++ features
Simplified constructor interfaces
Fully automated by compiler
Fully optimized by compiler
Generalizable feature
A true pleasure to use



Our Goal ***Realized!***

Runtime performance
Scoped allocator model
Localized ("arena") object memory
Entire-object placement in memory
Per-object metrics/measurement
Ubiquitous vocabulary types (handles)
Simple (to write/use) allocators
Rapid prototyping (e.g., `pmr` containers)
Predefined resources (e.g., `monotonic`)
Works seamlessly with *all* C++ features
Simplified constructor interfaces
Fully automated by compiler
Fully optimized by compiler
Generalizable feature
A true pleasure to use



Thank you!

References

- Pablo Halpern, *Allocators: the Good parts*, CppCon 2017, https://youtu.be/v3dz-AKOVL8
- John Lakos, *Local (Arena) Memory Allocators*, Meeting C++ 2017, Part I: https://youtu.be/ko6uywoC8ro, Part II: https://youtu.be/fN7nVzbRiEk
- Herb Sutter, *Meta: Thoughts on Generative C++*, CppCon 2017, https://youtu.be/4AfRAVcThyA
- Attila Fehér, *test allocator proposal and source code* https://github.com/bloomberg/p1160

Backup

Lambdas capture allocator awareness

- Lambda is allocator aware if it captures an allocator aware type

```
std::pmr2::vector<int> x = {2, 3, 5, 7, 11};
auto func = [x] using myAlloc (size_t i) { return x[i]; };

using Callback = std::pmr2::function<int(int)>;

std::pmr2::vector<Callback> v using myAlloc;
v.emplace_back([x] using myAlloc (size_t i) { return x[i]; } );
```

- Reference captures are never allocator aware
  - as reference types are never allocator aware

Allocating memory

- Mostly unchanged from C++20:

```
void MyContainer::push_back(const value_type& v)
  auto newNode = std::pmr2::get_allocator(*this).allocate_object<Node>(v);
  link_node(d_end, newNode);
}
```