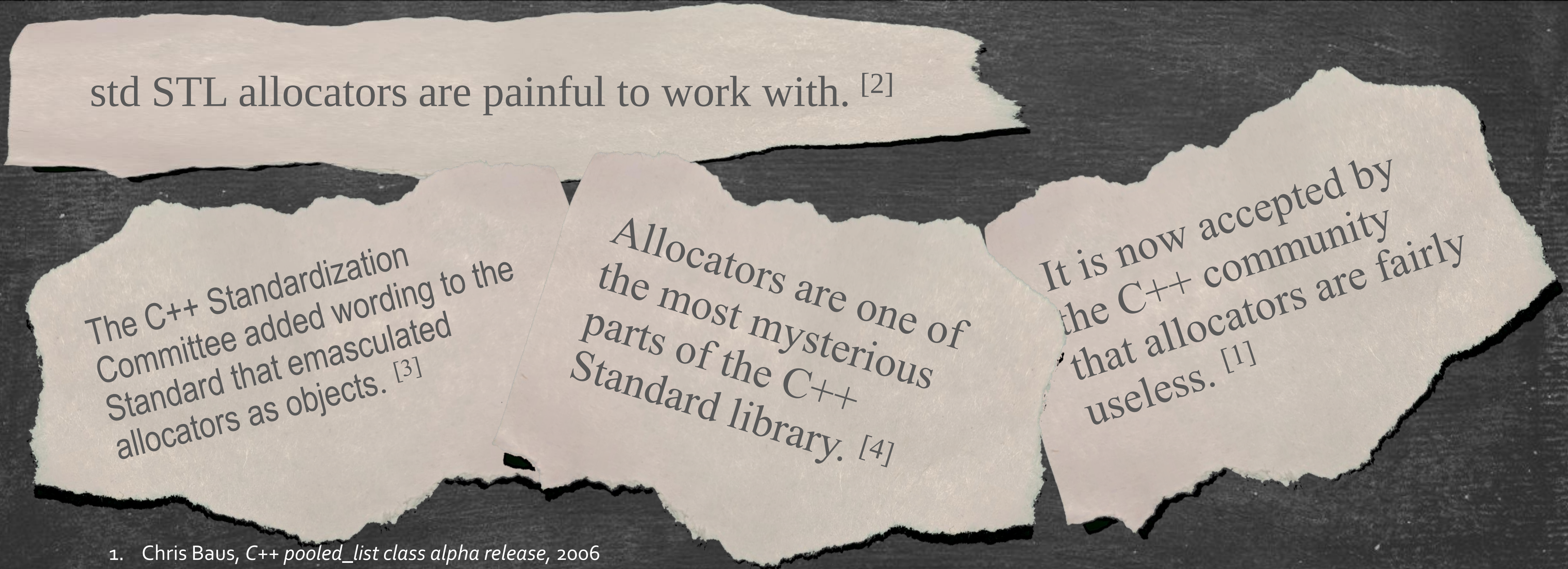
Allocators: the Good Parts

Pablo Halpern phalpern@halpernwightsoftware.com

CppCon 2017

Allocators: The reviews are not good



std STL allocators are painful to work with. [2]

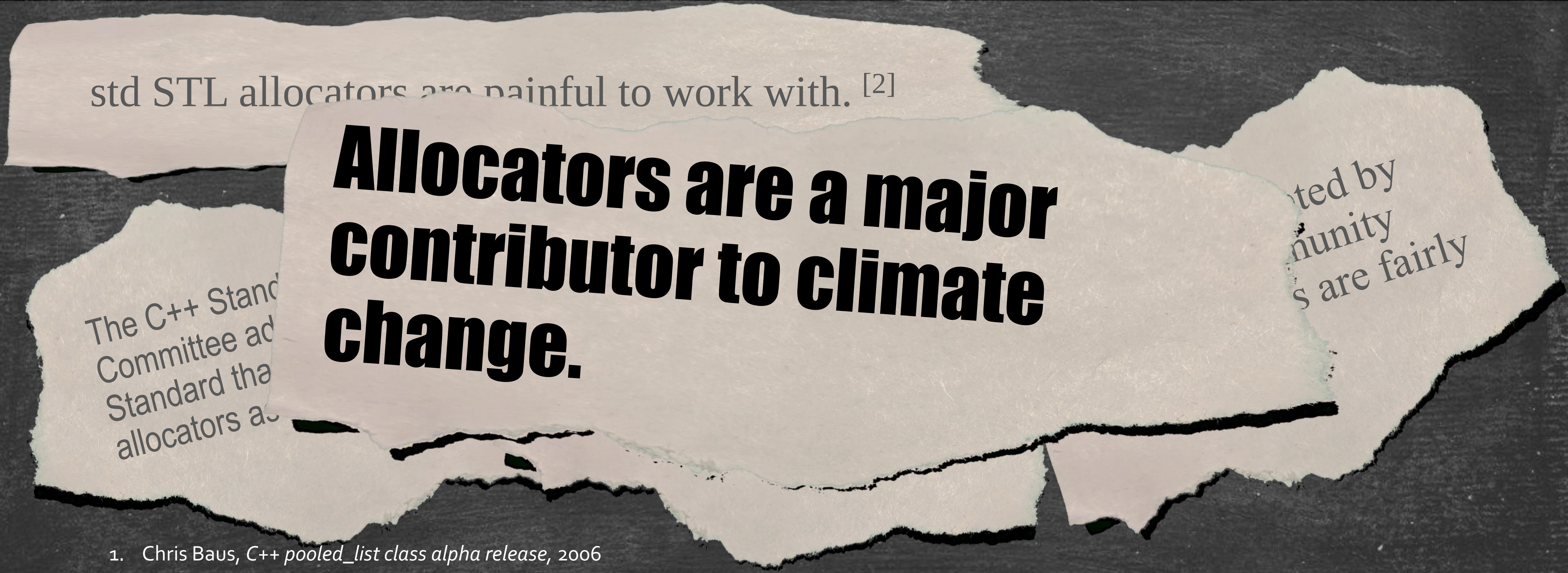
The C++ Standardization Committee added wording to the Standard that emasculated allocators as objects. [3]

Allocators are one of the most mysterious parts of the C++ Standard library. [4]

It is now accepted by the C++ community that allocators are fairly useless. [1]

1. Chris Baus, *C++ pooled_list class alpha release*, 2006
2. Paul Pedriana, *N2271 EASTL*, 2007
3. Meyers: *Effective C++ Digital Edition*, 2012
4. Matt Austern, *The Standard Librarian: What Are Allocators Good For?*, Dr. Dobbs, 2000

Allocators: The reviews are not good


std STL allocators are painful to work with. [2]

The C++ Stand
Committee ad
Standard tha
allocators a

Allocators are a major contributor to climate change.

ted by
nunity
s are fairly

1. Chris Baus, *C++ pooled_list class alpha release*, 2006
2. Paul Pedriana, *N2271 EASTL*, 2007
3. Meyers: *Effective C++ Digital Edition*, 2012
4. Matt Austern, *The Standard Librarian: What Are Allocators Good For?*, Dr. Dobbs, 2000

My Thesis

With a little guidance and few enhancements borrowed from C++17, allocators in C++11/14 become both useful and usable.

Contents

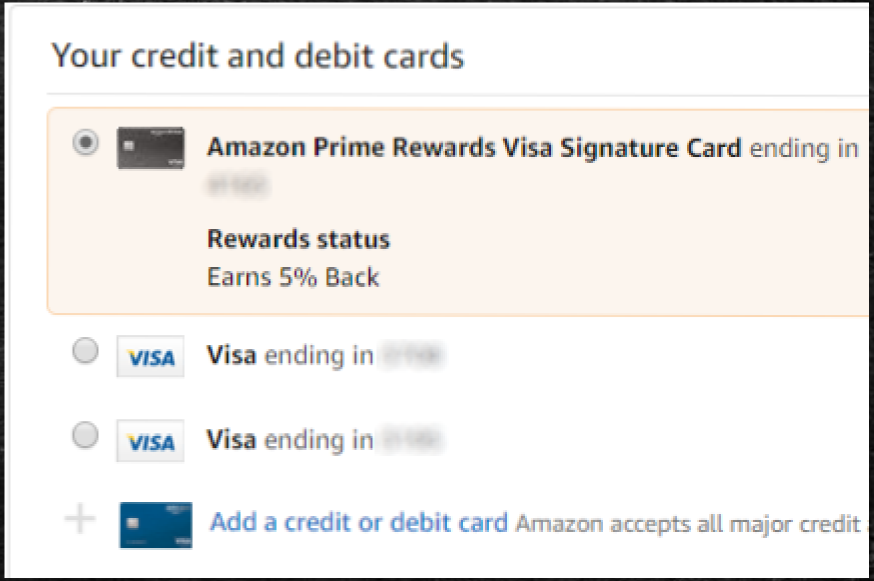
Goal: To learn the good parts of the allocator interface and best practices for using it.

- The "why" of allocators
- The core functionality of allocators
- That sounds simple, so where does the complexity come from?
- The simple allocator model, `pmr::polymorphic_allocator`
- Let's build a new memory resource type
- Let's build a new container type using the simple allocator model

The why of allocators

How do you want to pay?

- When you pay for a purchase, you *allocate* payment from a specific account.
  - Credit vs. debit
  - Personal vs. business
  - Joint vs. individual
- To the vendor, these are the same.
- To the purchaser, they are different – all money is not the same.
- There may be a default payment method.



Your credit and debit cards

[✓ **Amazon Prime Rewards Visa Signature Card** ending in [redacted]]

Rewards status
Earns 5% Back

Visa ending in [redacted]

Visa ending in [redacted]

Add a credit or debit card Amazon accepts all major credit

A user-supplied credit card lets the purchaser customize the transaction.

How do you want to allocate memory?

A user-supplied allocator is the credit card of memory allocation, because not all memory is the same. The user may wish to control

- The allocation algorithm for the expected use pattern.
- Locality of allocated memory
- Use of special-purpose memory (e.g., persistent memory)
- Thread locality (e.g., thread-specific memory)
- Statistics gathering or other instrumentation

The core functionality of allocators

An allocator is an object that provides the following two basic services:

- `allocate` – return a specified amount correctly-aligned memory for use by the client.
- `deallocate` – return the specified memory to the allocator for eventual reuse.

C++ takes this core functionality and adds many layers of complexity. Our job today is to strip away those layers and focus on the core functionality.

That sounds simple. Where does the complexity come from?

Complexity started at the beginning

- The original 1998 allocator model was incomplete and intended to handle a different set of problems (e.g., `near` and `far` pointers).
- Improvements made to the model in C++11 required extra complexity in order to retain backwards compatibility.
- The template-policy model leads to viral templatization of client code.
- To some extent, there was too much generalization in an attempt to please everyone.
- C++11 simplified the creation and use of allocators, but made implementation of containers harder.
  - STL containers always talk to the allocator using `allocator_traits`.
  - Propagation traits add flexibility but also complexity to the container.

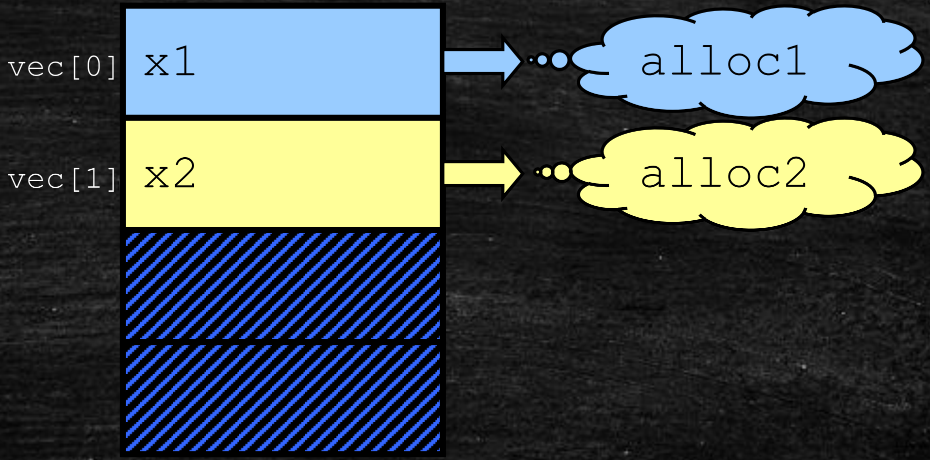
Layers of complexity in the standard

- Large numbers of `typedefs`
- The `rebind` template
- Generalized pointer model
- Allocator-type policies on every container
- `allocator_traits` for backwards compatibility.

Limiting ourselves to a useful subset of allocator features reduces complexity significantly.

The mess we inherited
Who’s got my allocator?

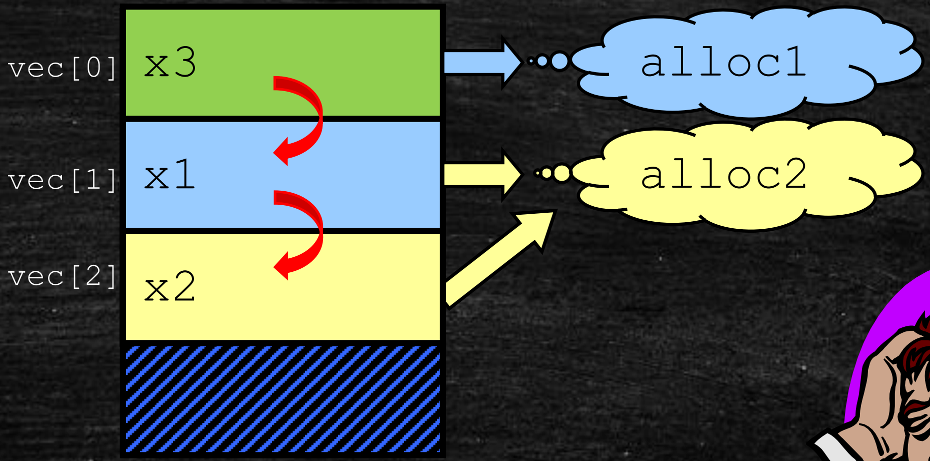
```
using CustomString =
  std::basic_string<char, char_traits<char>, CustomAlloc<char>>;
CustomAlloc<char> alloc1(SYSTEM), alloc2(LOCAL), alloc3;
CustomString x1(alloc1), x2(alloc2), x3(alloc3);
std::vector<CustomString> vec;

vec.push_back(x1);

vec.push_back(x2);

vec.reserve(4);
```



The mess we inherited

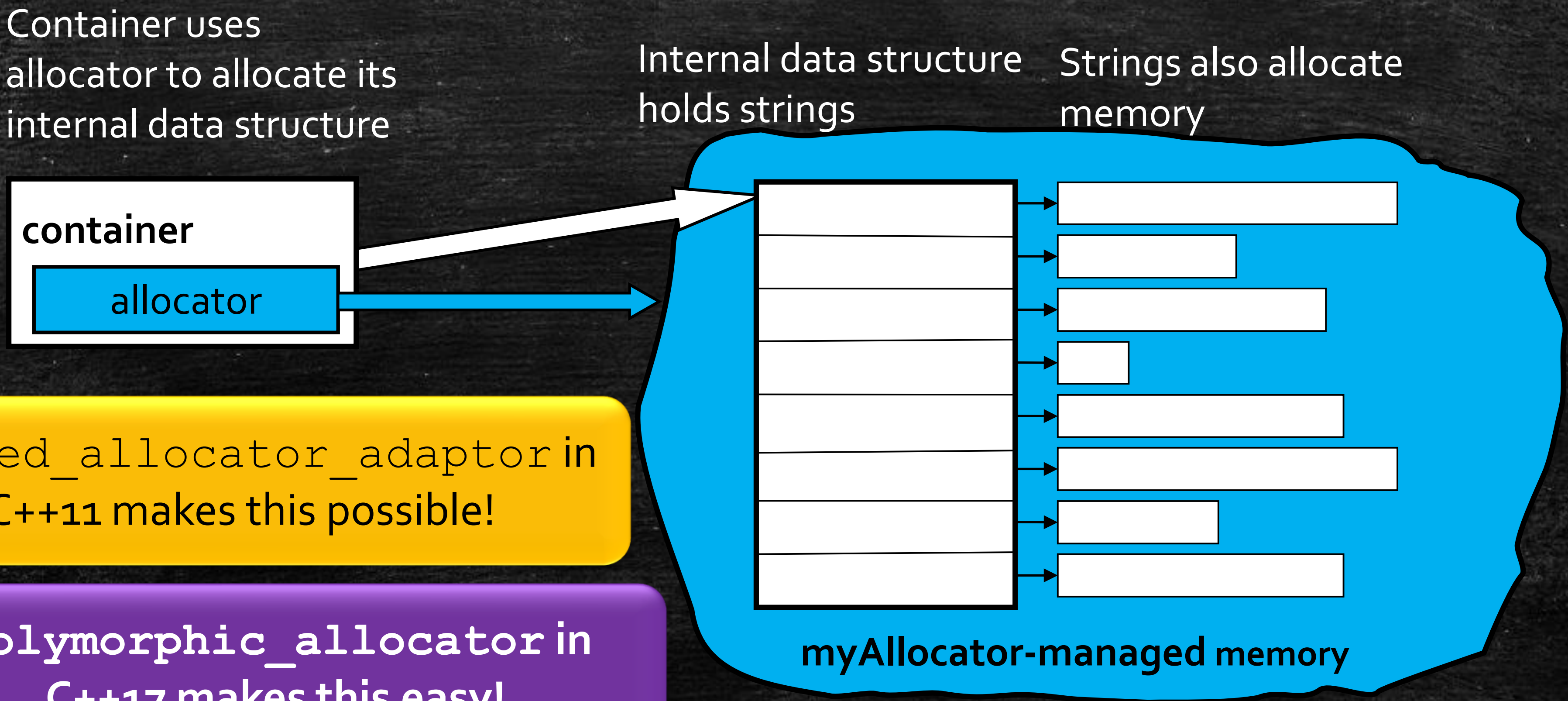
Who’s got my allocator?

```
using CustomString =
  std::basic_string<char, char_traits<char>, CustomAlloc<char>>;
CustomAlloc<char> alloc1(SYSTEM), alloc2(LOCAL), alloc3;
CustomString x1(alloc1), x2(alloc2), x3(alloc3);
std::vector<CustomString> vec;

vec.push_back(x1);

vec.push_back(x2);

vec.reserve(4);
vec.insert(vec.begin(), x3);
```



There has got to be a better way
The scoped allocator model

vector<string> container(myAllocator)


Container uses allocator to allocate its internal data structure
Internal data structure holds strings
Strings also allocate memory
container
allocator
myAllocator-managed memory

`scoped_allocator_adaptor` in C++11 makes this possible!

`polymorphic_allocator` in C++17 makes this easy!

The simple allocator model: `pmr::polymorphic_allocator`

C++17 supports a simpler allocator model

pmr = "polymorphic memory resource"

- `std::pmr::memory_resource` is a simple base class with `allocate` and `deallocate` member functions.
- `std::pmr::polymorphic_allocator` is a thin wrapper around a pointer to `pmr::memory_resource` for backwards-compatibility with the C++11 (and C++03) allocator model.
- `std::pmr::vector<T>` is an alias for `std::vector<T, std::pmr::polymorphic_allocator<T>>`. Similarly for the other allocator-aware standard containers.
- You don't have to wait for C++17 to use these! A C++11 version of these types is at https://github.com/phalpern/CppCon2017Code

Polymorphic memory resources

```
namespace std::pmr {

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

}
```

delegate to virtual functions

Polymorphic allocator

```
template <class Tp>
class polymorphic_allocator {
public:
   polymorphic_allocator() noexcept;
   polymorphic_allocator(memory_resource* r);
   memory_resource* resource() const;

   Tp* allocate(size_t n);
   void deallocate(Tp* p, size_t n);
   …
};
```

construct using default resource

convert from resource ptr

return resource ptr

Using `polymorphic_allocator<byte>` as the “one true allocator type”

- It is a single vocabulary type – no viral template explosion.
- It is a scoped allocator – standard containers will automatically pass the allocator to sub-objects.
- Byte allocation is directly available via the `resource()` member.
- Better than a raw pointer to `memory_resource` because:
  - It has a reasonable value when default-initialized.
  - It cannot be re-assigned by accident.
  - It plays nice with the rest of the standard library.

Allocators got a whole lot easier

Task	C++98/C++03	C++11/C++14	C++17 polymorphic_ allocator<byte>
Use an allocator	**MEDIUM** viral templates	**MEDIUM** viral templates	**EASY**
Create an allocator	**MEDIUM** Lots of boilerplate, non-portable state	**EASY**	**EASY** just derive from memory_resource
Create a scoped allocator	**IMPOSSIBLE**	**MEDIUM-EASY** alias scoped_ allocator_adaptor	**EASY** polymorphic_allocator is scoped
Create a new allocator-aware container	**MEDIUM** rebinding needed, ignore allocator state?	**HARD** propagation traits, allocator_traits	**EASY** skip C++11 complexity

Let’s build a new memory resource type

Wait! Maybe there's an app for that!

C++17 Defines several standard resources:

- **`new_delete_resource()`**: Allocates using `::operator new`
- **`null_memory_resource()`**: Throws on allocation
- **`synchronized_pool_resource`**: Thread-safe pools of similar-sized memory blocks.
- **`unsynchronized_pool_resource`**: Non-thread-safe pools of similar-sized memory blocks.
- **`monotonic_buffer_resource`**: Super-fast, non-thread-safe allocation into a buffer with do-nothing deallocation.

A memory resource for testing

We will build a new memory resource for testing purposes which will

- Keep track of amount of how much memory is allocated, deallocated, and a high-water mark for memory outstanding.
- Check that every deallocation matches an allocation.
- Check for memory leaks – any memory still allocated when the resource's destructor is called is considered leaked.

Full source available at: https://github.com/phalpern/CppCon2017Code

Authoring a test resource
public interface

```
class test_resource : public pmr::memory_resource
{
public:
  explicit test_resource(pmr::memory_resource *parent =
                           pmr::get_default_resource());
  ~test_resource();

  pmr::memory_resource *parent() const;

  size_t bytes_allocated() const;
  size_t bytes_deallocated() const;
  size_t bytes_outstanding() const;
  size_t bytes_highwater() const;
  size_t blocks_outstanding() const;

  static size_t leaked_bytes();
  static size_t leaked_blocks();
  static void   clear_leaked();
  …
};
```

Authoring a test resource
virtual function overrides

```
class test_resource : public pmr::memory_resource
{
  …
protected:
  void *do_allocate(size_t bytes, size_t alignment) override;
  void do_deallocate(void *p, size_t bytes,
                     size_t alignment) override;
  bool do_is_equal(const pmr::memory_resource& other)
                                        const noexcept override;
  …
};
```

Authoring a test resource
private allocation record type

```
class test_resource : public pmr::memory_resource
{
  …
private:
  // Record holding the results of an allocation
  struct allocation_rec {
    void   *m_ptr;
    size_t  m_bytes;
    size_t  m_alignment;
  };
  …
};
```

Authoring a test resource
private data members

```
class test_resource : public pmr::memory_resource
{
  …
private:
  …
  pmr::memory_resource          *m_parent;
  size_t                         m_bytes_allocated;
  size_t                         m_bytes_outstanding;
  size_t                         m_bytes_highwater;
  pmr::vector<allocation_rec>    m_blocks;

  static size_t                  s_leaked_bytes;
  static size_t                  s_leaked_blocks;
};
```

Authoring a test resource
Allocating memory

```
void *test_resource::do_allocate(size_t bytes,
                                 size_t alignment) {

  void *ret = m_parent->allocate(bytes, alignment);
  m_blocks.push_back(allocation_rec{ret, bytes, alignment});
  m_bytes_allocated   += bytes;
  m_bytes_outstanding += bytes;
  if (m_bytes_outstanding > m_bytes_highwater)
    m_bytes_highwater = m_bytes_outstanding;

  return ret;
}
```

Authoring a test resource

Deallocating memory

```
void test_resource::do_deallocate(void *p, size_t bytes,
                                   size_t alignment) {
  // Check that deallocation args exactly match allocation args.
  auto i = std::find_if(m_blocks.begin(), m_blocks.end(),
                        [p](allocation_rec& r){
                          return r.m_ptr == p; });
  if (i == m_blocks.end())
    throw std::invalid_argument("deallocate: Invalid pointer");
  else if (i->m_bytes != bytes)
    throw std::invalid_argument("deallocate: size mismatch");
  else if (i->m_alignment != alignment)
    throw std::invalid_argument("deallocate: Alignment mismatch");

  m_parent->deallocate(p, i->m_bytes, i->m_alignment);
  m_blocks.erase(i);
  m_bytes_outstanding -= bytes;
}
```

Authoring a test resource

Test for equality

```
bool test_resource::do_is_equal(const pmr::memory_resource& other)
                                                     const noexcept {
   // Two test resources are equal if they are the same resource.
   return this == &other;
}
```

Getting it right – memory resources

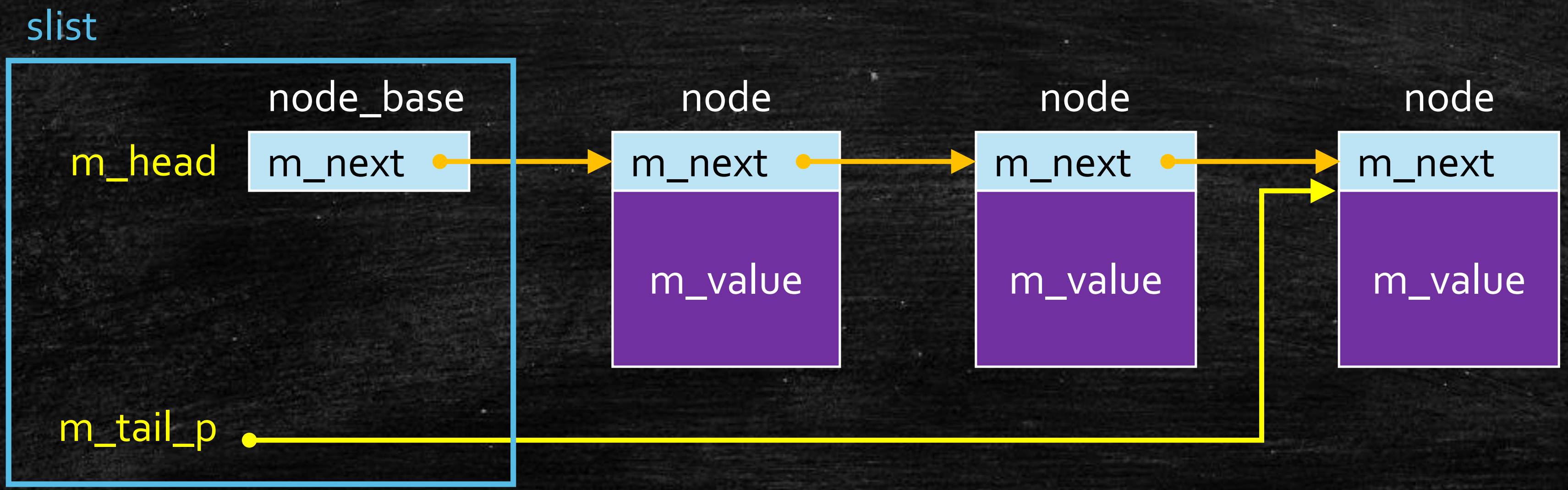
- Creating a new memory resource is as easy as deriving from `pmr::memory_resource`.
- Override `do_allocate()` and `do_deallocate()` to do the real work.
- Override `do_is_equal()` to test for equality.
  - In most cases `return this == &other` (identity equality) is sufficient
  - If there is no resource-specific state, `return true` is correct.
  - In rare cases, something more complex is needed.

Let’s build a new container type using the simple allocator model

A linked-list container type – slist<Tp>

`slist<Tp>` is a sequence container like `list<Tp>` and `vector<Tp>`

- Supports push_back/front, emplace_back/front, front, emplace, insert, erase, pop_front, begin, end, size, empty, swap
- Implemented as a singly-linked list with a sentinel at the head:



Authoring a container – slist<Tp>

Node types

```
template <typename Tp> struct node;

template <typename Tp>
struct node_base {           // Base class holds no value
  node<Tp> *m_next;

  node_base() : m_next(nullptr) { }
  node_base(const node_base&) = delete;
  node_base operator=(const node_base&) = delete;
};

template <typename Tp>
struct node : node_base<Tp> {   // Derived class holds value
  union U {
    Tp  m_value;
    constexpr U(){}
  } u;
};
```

Value nested in union suppresses constructor invocation

Authoring a container – slist<Tp>

Private data members

```
template <typename Tp>
class slist {
  …
private:
  …
  node_base<Tp>   m_head;          // sentinel node
  node_base<Tp>  *m_tail_p;        // pointer to last node
  size_t          m_size;
  allocator_type  m_allocator;
};
```

Authoring a container – slist<Tp>

Types and constructors

```
template <typename Tp>
class slist {
public:
  using value_type      = Tp;
  using reference       = value_type&;
  using const_reference = value_type const&;
  using difference_type = std::ptrdiff_t;
  using size_type       = std::size_t;
  using allocator_type  = pmr::polymorphic_allocator<byte>;
  using iterator        = …;
  using const_iterator  = …;

  // Constructors
  slist(allocator_type a = {})
    : m_head(), m_tail_p(&m_head), m_size(0), m_allocator(a) {}
  slist(const slist& other, allocator_type a = {});
  slist(slist&& other);
  slist(slist&& other, allocator_type a);
```

cut-and-paste boilerplate

Non-template use of polymorphic_allocator

Every constructor has a variant taking an allocator

Authoring a container – slist<Tp>
inserting an element

```
template <typename Tp>
template <typename... Args>
typename slist<Tp>::iterator
slist<Tp>::emplace(iterator i, Args&&... args) {
  node* new_node = static_cast<node*>(
    m_allocator.resource()->allocate(sizeof(node), alignof(node)));
  try { m_allocator.construct(std::addressof(new_node->u.m_value),
                              std::forward<Args>(args)...); }
  catch (...) { m_allocator.resource()->deallocate(new_node,
                                          sizeof(node), alignof(node));
    throw; }

  new_node->m_next = i.m_prev->m_next;
  i.m_prev->m_next = new_node;
  if (i.m_prev == m_tail_p)
    m_tail_p = new_node;  // Added at end
  ++m_size;
  return i;
}
```

Secret sauce #1:
Allocate a node, then construct a value within it.

Authoring a container – slist<Tp>

erasing an element

```
template <typename Tp>
typename slist<Tp>::iterator
slist<Tp>::erase(iterator b, iterator e) {
  node *erase_next = b.m_prev->m_next;
  node *erase_past = e.m_prev->m_next; // one past last erasure
  if (nullptr == erase_past)
    m_tail_p = b.m_prev;  // Erasing at tail
  b.m_prev->m_next = erase_past; // splice out sublist
  while (erase_next != erase_past) {
    node* old_node = erase_next;
    erase_next = erase_next->m_next;
    --m_size;
    m_allocator.destroy(std::addressof(old_node->u.m_value));
    m_allocator.resource()->deallocate(old_node,
                                       sizeof(node), alignof(node));
  }

  return b;
}
```

Secret sauce #2
Destroy value within node, then deallocate the node.

Authoring a container – slist<Tp>

copy and move assignment

```
template <typename Tp>
slist<Tp>& slist<Tp>::operator=(const slist& other) {
  if (&other == this) return *this;
  erase(begin(), end());
  for (const Tp& v : other)
    push_back(v);
  return *this;
}

template <typename Tp>
slist<Tp>& slist<Tp>::operator=(slist&& other) {
  if (m_allocator == other.m_allocator)
    swap(other); // non-copying move
  else
    operator=(other);  // Copy assign
  return *this;
}
```

Don't move the allocator. Never change the allocator of an existing object!

Authoring a container – slist<Tp>
copy and move construction

```
template <typename Tp>
slist<Tp>::slist(const slist& other, allocator_type a)
  : slist(a) {
  operator=(other);
}
```

Do not propagate other allocator on copy construction

```
template <typename Tp>
slist<Tp>::slist(slist&& other)
  : slist(other.get_allocator()) {
  operator=(std::move(other));
}
```

Always propagate other allocator on regular move construction

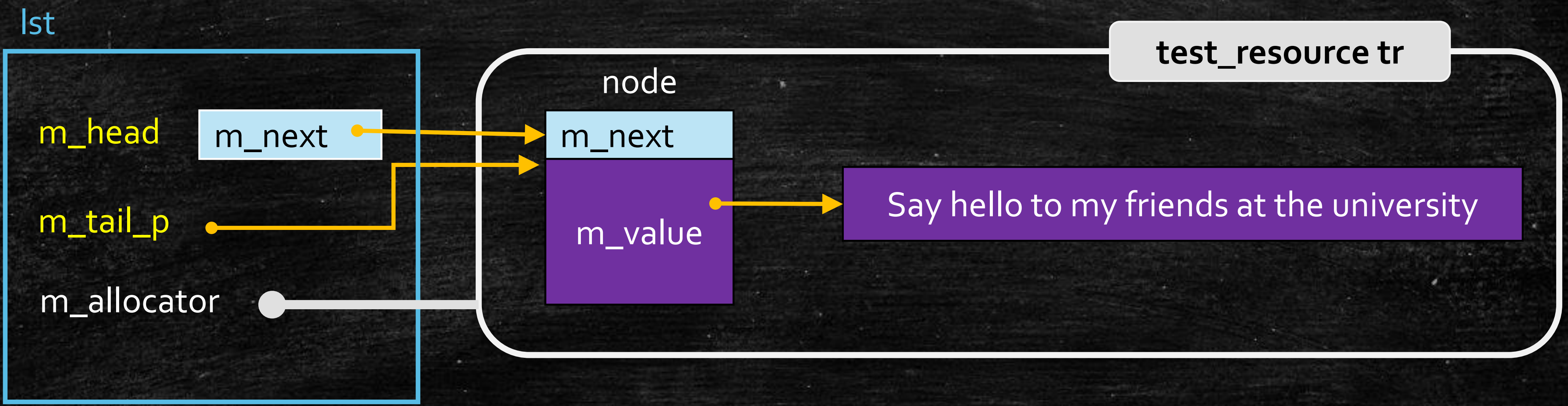
```
template <typename Tp>
slist<Tp>::slist(slist&& other, allocator_type a)
  : slist(a) {
  operator=(std::move(other));
}
```

Do not propagate other allocator on *extended* move construction

Putting it all together

Automatic conversion from `memory_resource*` to `polymorphic_allocator`

```
test_resource tr;
slist<pmr::string> lst(&tr);
pmr::string hello = "Say hello to my friends at the university";
assert(0 == tr.bytes_allocated());  // No use of tr yet
lst.push_back(hello);
assert(2 == tr.blocks_outstanding()); // 2 blocks allocated
```



Getting it right – constructors

- *Every* constructor should have a variant that takes a polymorphic_allocator<byte>:

```
struct X {
  using allocator_type = polymorphic_allocator<byte>;
  X(const X&, allocator_type = {});

  // Allocator at start of arg list
  template <class... Args> X(Args&&...);
  template <class... Args> X(allocator_arg_t, allocator_type,
                              Args&&...);
```

- The move constructor cannot use a default allocator:
  - `X(X&& other)` stores `other.get_allocator()`.
  - `X(X&& other, allocator_type a)` stores `a`.

Getting it right – copy and move

- Copy construction and copy assignment are require no special techniques.
- For move assignment, check for allocator equality. If not equal *copy* instead of *move* the elements. Never replace the allocator of an existing object!

```
X& operator=(X&& other) {
   if (get_allocator() != other.get_allocator())
      this->operator=(other);   // Invoke copy-assignment
   else
      // do move
```

- Similarly, `X(X&& v, allocator_type a)`, *extended move constructor*, must copy if `a != v.get_allocator()`
  - The easiest way is to construct using `a`, then delegate the test, move, copy to the move-assignment operator

Getting it right – adding elements

- For node-based containers, define a node containing an element that is not automatically constructed.
  - Putting the element inside a union will suppress the element constructor.

  Allocate raw memory for nodes using `allocator.resource()->allocate(bytes, alignment)`, where bytes and alignment are typically the size and alignment of your node type:

```
node* new_node = static_cast<node*>(
    allocator.resource()->allocate(sizeof(node), alignof(node)));
```

- Construct new elements within nodes using `get_allocator().construct()`:

```
allocator.construct(std::addressof(new_node->m_value),
                    std::forward<Args>(args)...);
```

Simplifications over C++11

The consistent use of `polymorphic_allocator<byte>` is much simpler than the old allocator model:

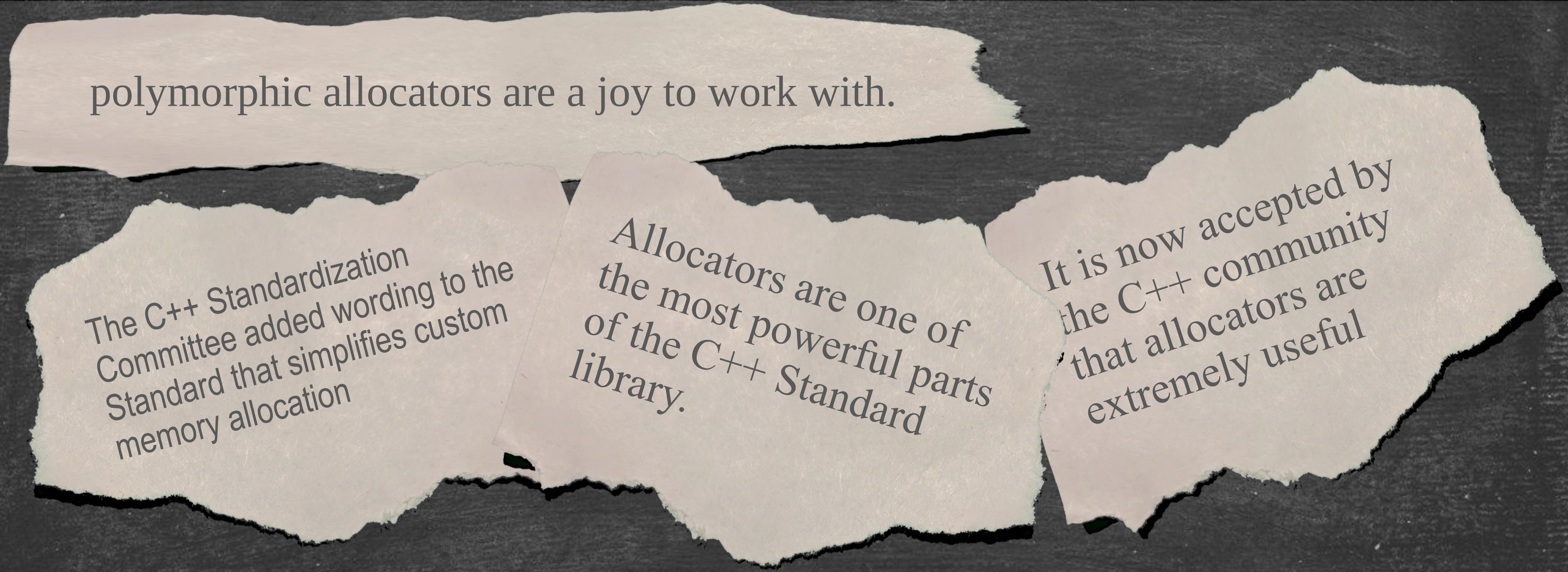
- No allocator template argument – allocator is always the same.
- No need for `allocator_traits` – every allocator has the same traits.
- No propagation traits – allocators don’t propagate except on move construction.
- No rebind – just allocate bytes
- Simple, understandable defaults.

Questions?

Conclusions

- The C++ allocator model is complex for historical reasons, but it doesn't have to be.
- C++11 and C++17 support significantly-better allocator models
- The trick is to leave the old complexity behind and consistently use the new models.
- Use `pmr::polymorphic_allocator<byte>` as a vocabulary type.
- Derive from pmr::memory_resource to create new allocation mechanisms.

Rave reviews for polymorphic allocators!



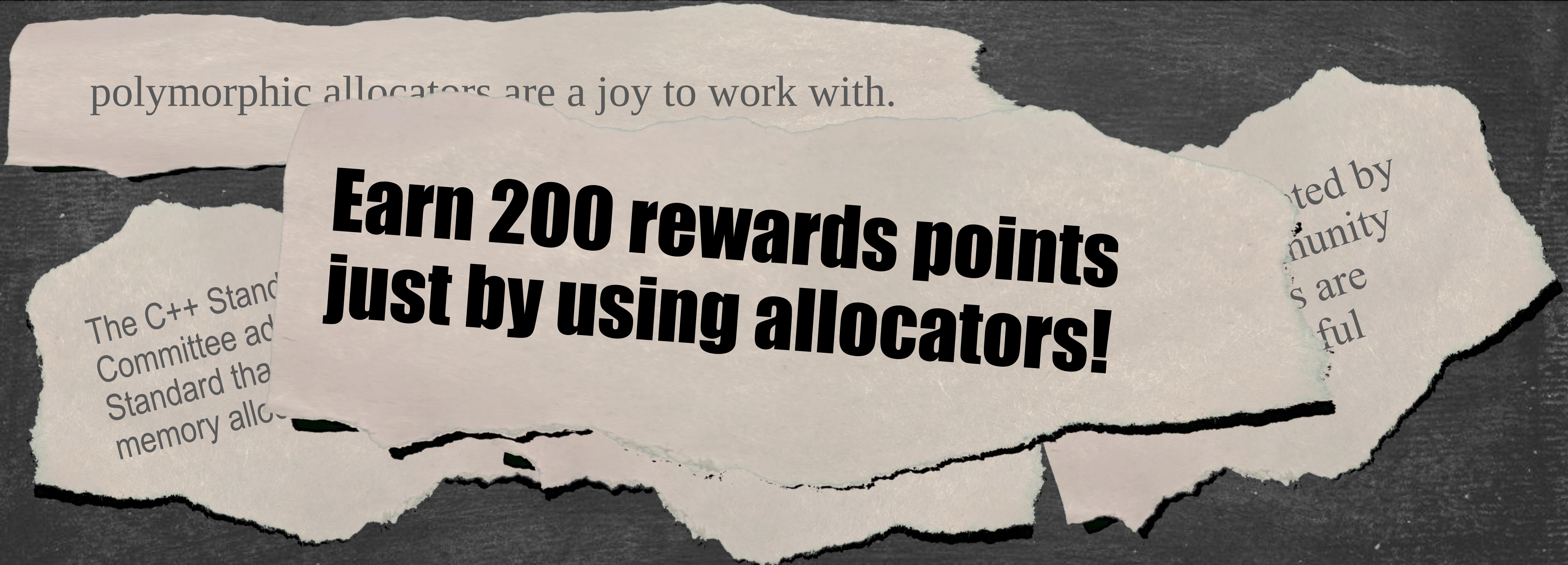
polymorphic allocators are a joy to work with.

The C++ Standardization Committee added wording to the Standard that simplifies custom memory allocation

Allocators are one of the most powerful parts of the C++ Standard library.

It is now accepted by the C++ community that allocators are extremely useful

Rave reviews for polymorphic allocators!



polymorphic allocators are a joy to work with.

Earn 200 rewards points just by using allocators!

The C++ Stand
Committee ad
Standard tha
memory allo

ted by
nunity
s are
ful