How to Write a Custom Allocator

Bob Steagall

CppCon 2017

Guidelines for How to Write a Custom Allocator

Bob Steagall
CppCon 2017

On Writing Allocators

We've all heard heroic tales of <u>other</u> people doing this.

- John McFarlane, CppCon 2017

Overview

- What is an allocator and why write your own?
- Some important background
- Modern allocator facilities
- A container’s perspective
- A few guidelines

What is an Allocator?

The basic purpose of an allocator is to provide a source of memory for a given type, and a place to return that memory to once it is no longer needed.

- Bjarne Stroustrup, TCPL, 4th Edition

A service that grants exclusive use of a region of memory to a client.

- Alisdair Meredith, CppCon 2017

What is a Standard Library Allocator?

- A kind of object used by the C++ standard library to manage memory
  - Implementation detail of standard containers, as a template parameter

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

- Invented by Alexander Stepanov as part of the original STL
  - Attempt to solve near/far/huge pointer types on PCs at the time
  - Effort to the library more flexible and independent of the underlying memory model

- Containers have special requirements
  - Need an interface that is more “granular” than `new` and `delete`
  - Needed finer control

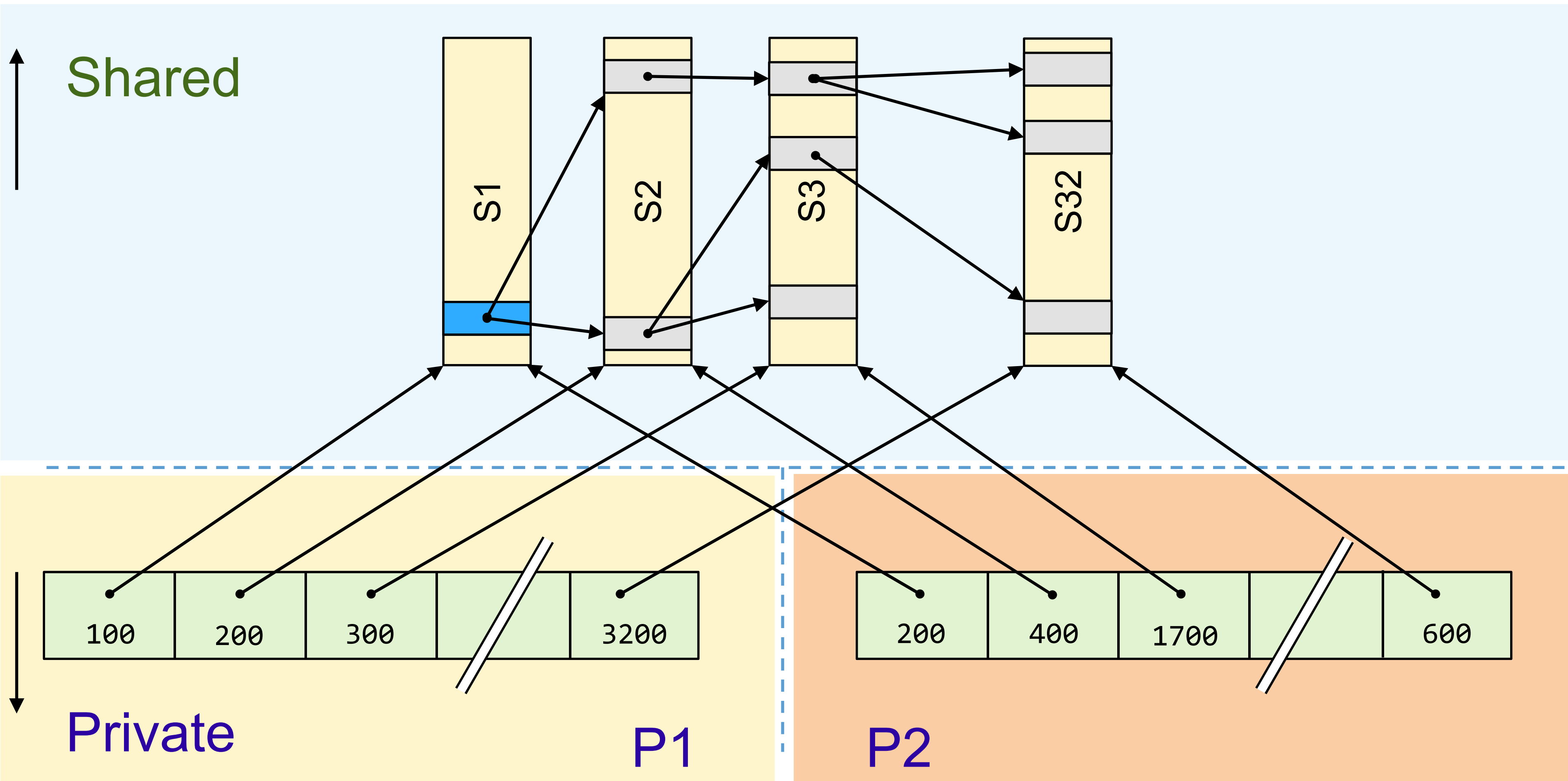
Allocator Mission

- Perform allocation / construction / destruction / deallocation services
  - Separate allocation from construction
  - Separate destruction from deallocation
- Encapsulate information about allocation strategy
- Encapsulate information about addressing model
- Hide memory management and addressing model details from containers
- Support reuse of allocation strategies across container types

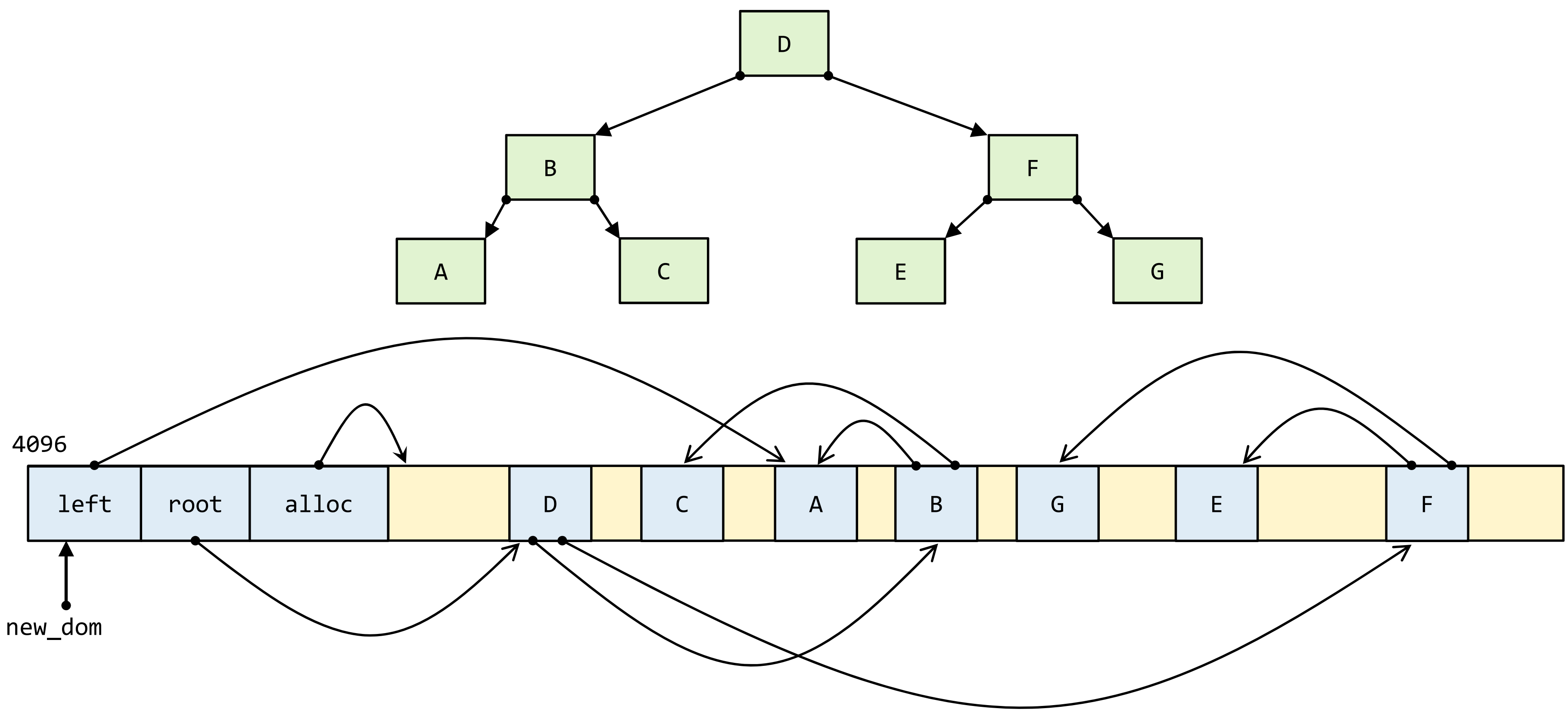
Why Write Your Own Allocator?

- Higher performance
  - Stack-based allocation
  - Per-container private allocation
  - Per-thread allocation
  - Pooled / slab allocation
  - Arena allocation
- Debug / instrumented / test
- Relocatable data
  - Shared memory
  - Self-contained heaps

Motivating Example – Shared Memory Data Structures

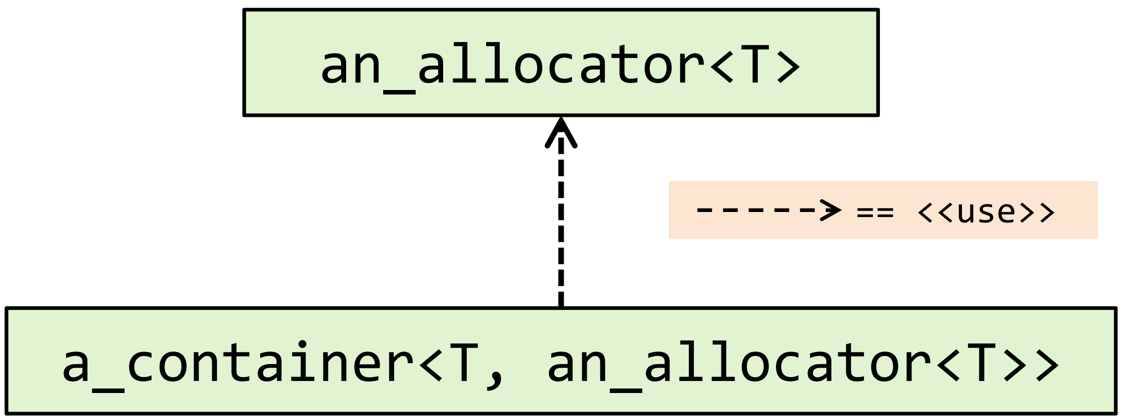


Motivating Example – Self-Contained DOM



A Brief Allocator History

C++03 Allocators



C++03 Allocators

- Containers obtained their allocation services directly from their allocator template argument:

```
T* p = my_alloc.allocate(1u);
```

- Containers were free to assume that:

```
using pointer            = T*;
using const_pointer      = T const*;
using void_pointer       = void*;
using const_void_pointer = void const*;
using size_type          = size_t;
using difference_type    = ptrdiff_t;

my_allocator<Foo>   a, b;  (a == b) == true;
b.deallocate(a.allocate(1u));   //- Well-defined
```

C++03 Default Allocator

```
template<class T> struct allocator
{
    typedef size_t     size_type;
    typedef ptfdiff_t  difference_type;
    typedef T*         pointer;
    typedef T const*   const_pointer;
    typedef T&         reference;
    typedef T const&   const_reference;
    typedef T          value_type;

    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);
};

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++03 Default Allocator – Allocation/Deallocation

```
template<class T> inline T*
allocator<T>::allocate(size_t n)
{
    return static_cast<T*>(::operator new(n * sizeof(T)));
}


template<class T> inline void
allocator<T>::deallocate(T* p, size_t)
{
    ::operator delete(p);
}
```

C++03 Default Allocator – Construction/Destruction

```
template<class T> inline void
allocator<T>::construct(pointer p, T const& val)
{
    ::new((void*)p) T(val);
}


template<class T> inline void
allocator<T>::destroy(pointer p)
{
    return ((T*)p)->~T();
}
```

C++03 Default Allocator – Comparison

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

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

C++03 Default Allocator

```
template<class T> struct allocator
{
    typedef size_t     size_type;
    typedef ptfdiff_t  difference_type;
    typedef T*         pointer;
    typedef T const*   const_pointer;
    typedef T&         reference;
    typedef T const&   const_reference;
    typedef T          value_type;

    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);
};

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++03 Default Allocator – Rebinding

```
template<class T, class Alloc = allocator<T>>
class list
{
     ...
     typedef typename Alloc::template rebind<list_node<T>>::other  node_allocator;

     typedef typename node_allocator::pointer  node_pointer;
     ...
};


template<class T, class Alloc>  typename list<T,Alloc>::node_pointer
list<T,Alloc>::alloc_node(T const& t)
{
    node_allocator  na(this->m_alloc);
    node_pointer    np = na.allocate(1u);
    na.construct(np, t);
    return np;
}
```

C++03 Allocators – Implications

- Implementations assumed that pointers were always `T*`
  - No support for synthetic pointers / alternate addressing models

- Implementations assumed that instances are always equal and interchangeable
  - Standard containers did not support stateful allocators

- Many interesting problems could not be solved using the standard containers
  - For example: shared memory data structures, self-contained heaps

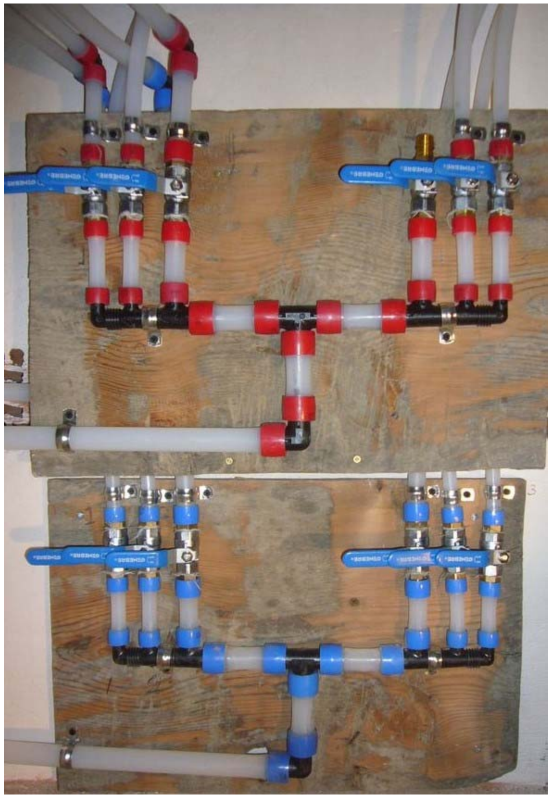
- Scoped allocation was tricky
  - Consider the case of `map<string, vector<list<string>>>`

Modern C++ Allocator Facilities

Modern Allocator Plumbing

- At first, it may seem like this…

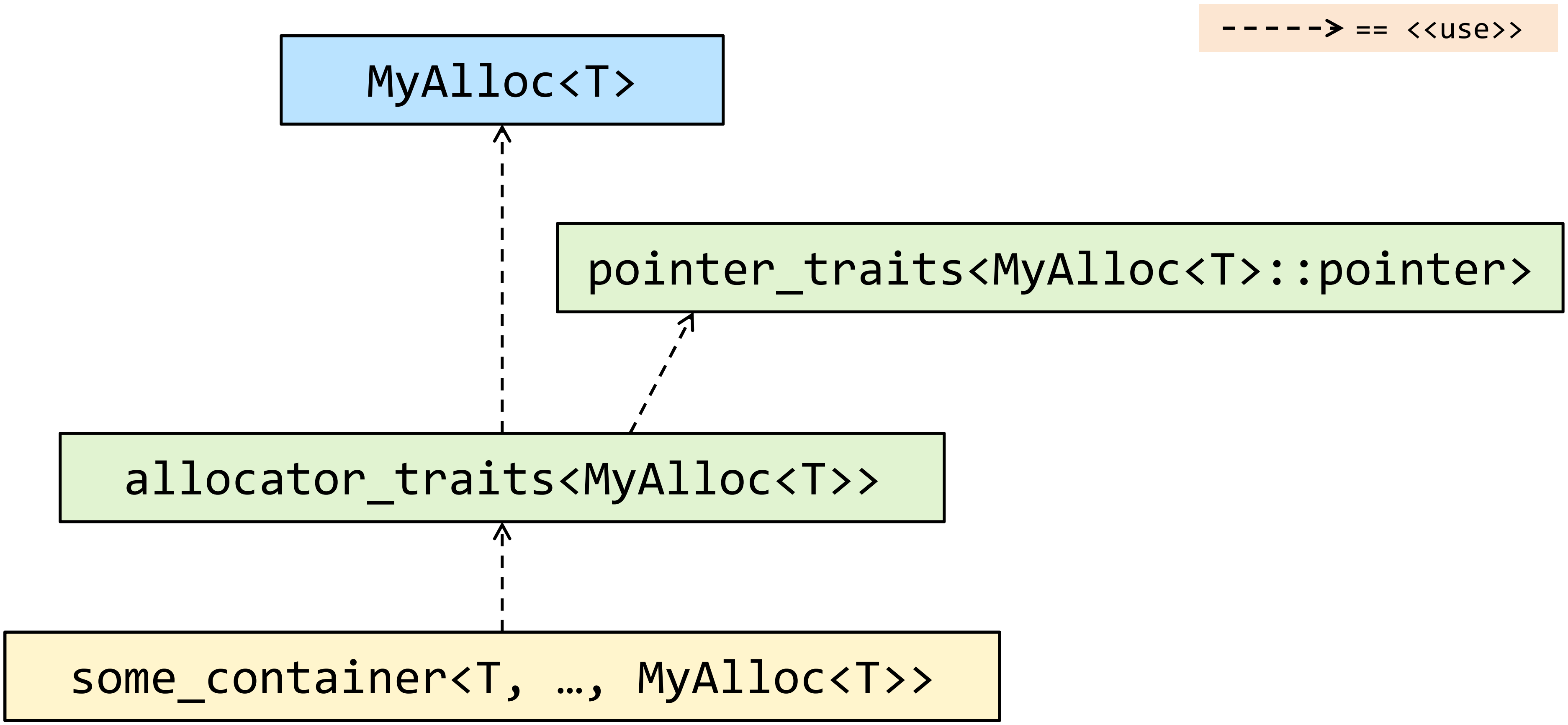
- But really, it's closer to this…



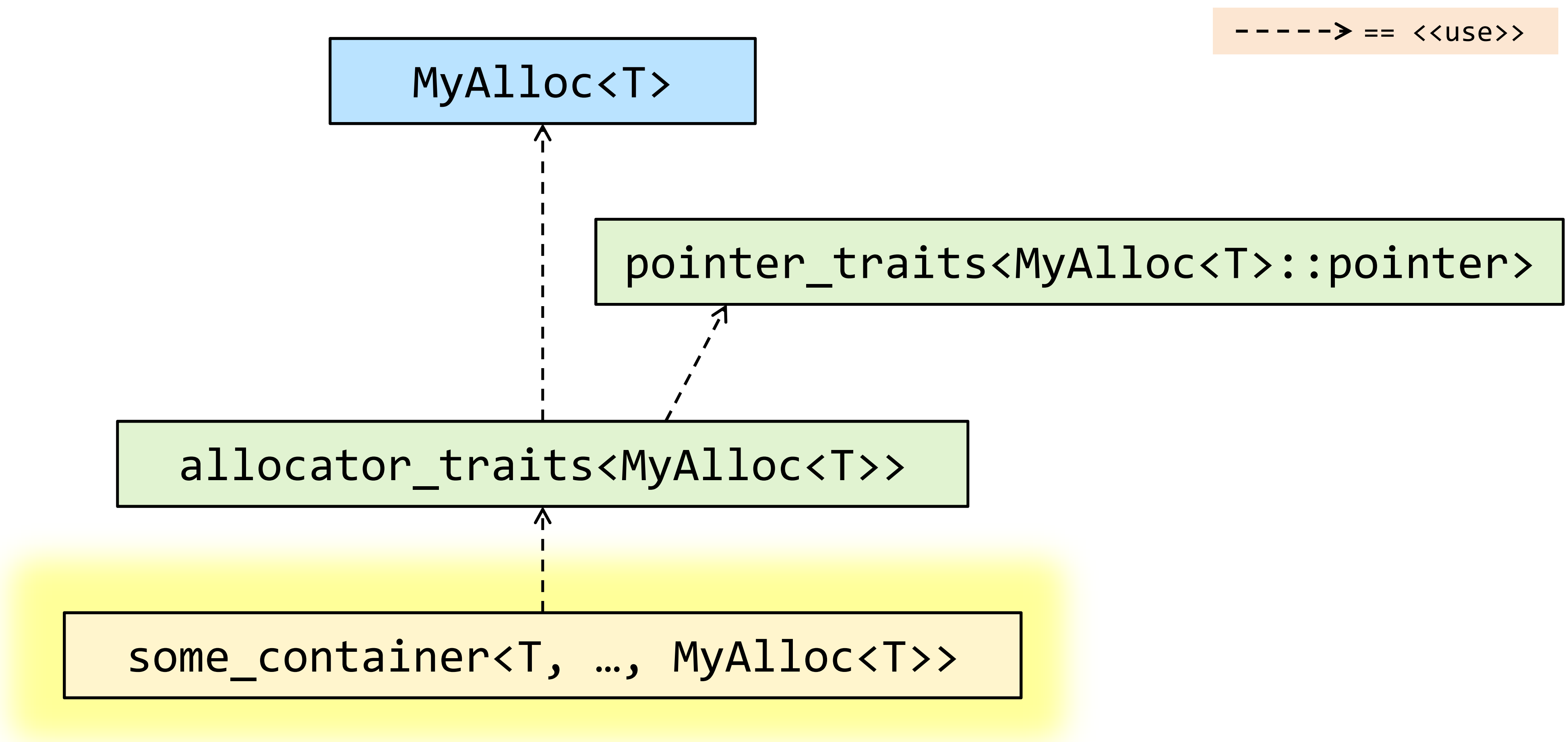
New Requirements to Improve Allocator Support in Modern C++

- *nullablepointer.requirements*
  - Pointer-like type that supports null values
- *allocator.requirements*
  - Defines what allocator is and its relationship to allocator traits
- *pointer.traits*
  - Describes a uniform interface to pointer-like types used by `allocator_traits`
- *allocator.traits*
  - Describes uniform interface to allocator types used by containers
- *allocator.adaptor*
  - Describes an adaptor template that supports deep propagation of allocators.
- *container.requirements.general*
  - Defines allocator-aware container requirements

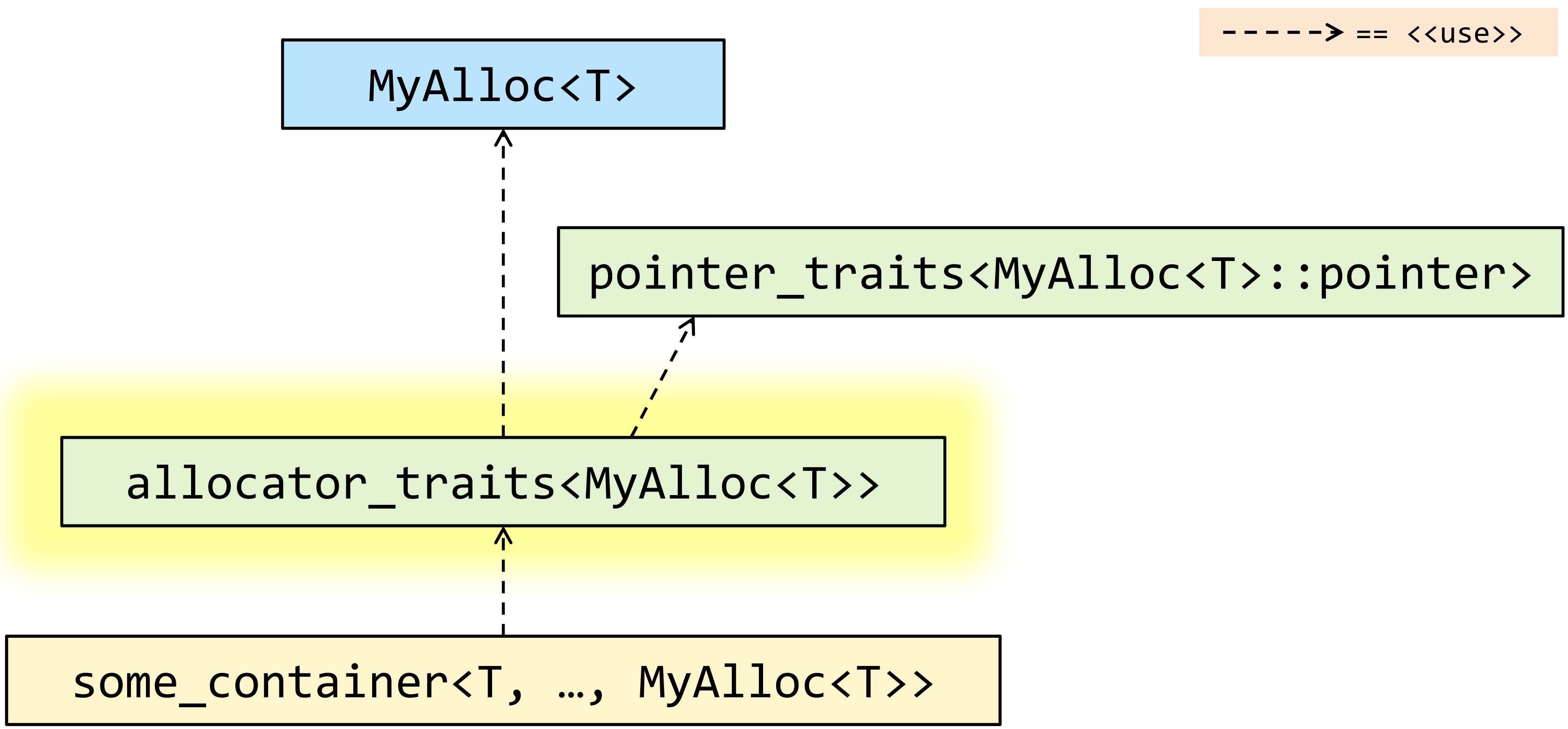
Modern C++ Allocator Model



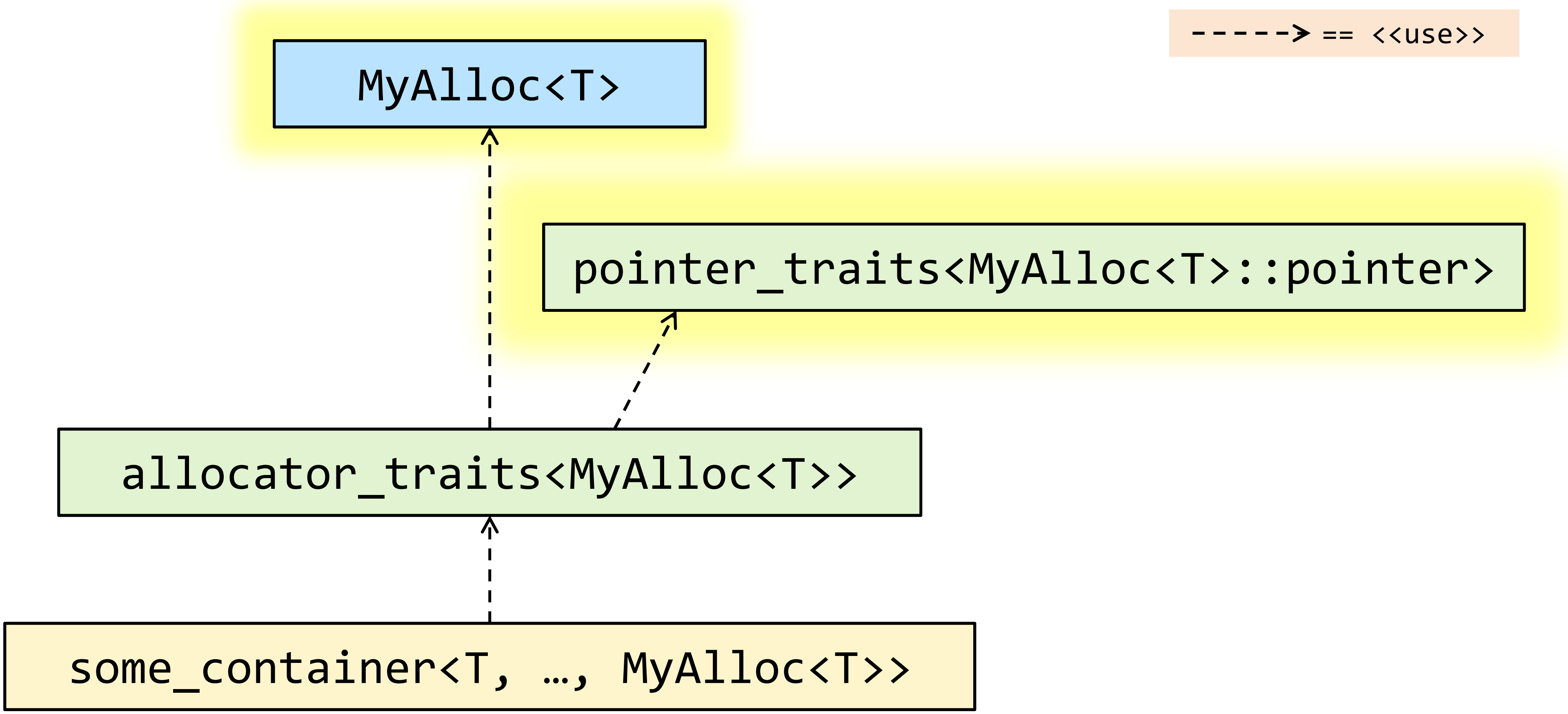
Modern C++ Allocator Model



Modern C++ Allocator Model



Modern C++ Allocator Model



Allocator Awareness

- What is it?
  - Pedantically – fulfilling the requirements in Table 86 (see N4687)

- *Allocator-awareness* means *doing something that makes sense* when a container uses a non-default allocator

- In modern C++, allocators
  - Can employ synthetic pointers (aka pointer-like types) for addressing operations
  - Can be stateful
  - Can compare non-equal and be non-interchangeable
  - Can have non-trivial copy/move/swap semantics (propagate)

A Quick Word From Our Sponsors About Allocator Propagation

- Lateral propagation refers to what happens to a container's allocator during
  - Copy/move construction
  - Copy/move assignment
  - Swap

- Deep propagation refers to "nesting" the allocator of the outermost container in a container hierarchy
  - Consider once more `map<string, vector<list<string>>>`
  - Modern C++ has `scoped_allocator_adaptor` to help with this situation

- Today's focus will be on lateral propagation

Implications of Allocator Awareness for **Library Implementors**

- Containers must now perform all allocator-related operations using `allocator_traits`
- Containers must support the case of non-equal allocators
- Containers must support lateral propagation
- Containers now include allocator-extended constructors to support deep propagation
- Containers no longer use `reference` or `const_reference` nested type aliases from allocators
- `allocator_traits::construct()` uses perfect forwarding
- `allocator_traits::construct()`/`destroy()` now use `T*` instead of the `pointer` type alias

Implications of Allocator Awareness for **Library Users**

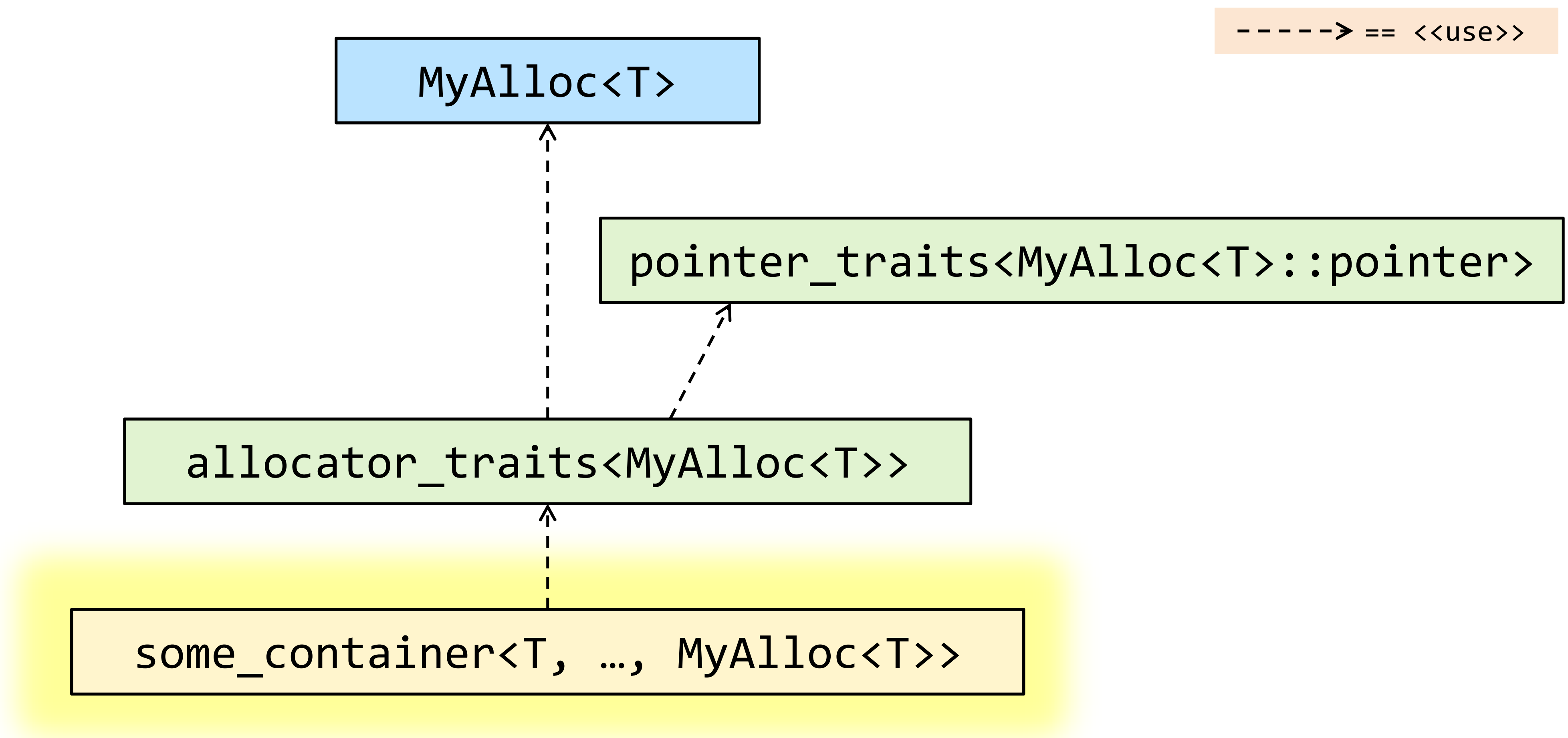
- Stateful allocators can be used with the standard containers
- It is now (relatively) safe and straightforward to use scoped allocators for nested container hierarchies
- Some mutating container operations may lose `noexcept` property (depending on implementation)

Implications of Allocator Awareness for **Allocator Implementors**

- Some of the previous verbiage and burden is removed
  - No need to define `reference` or `const_reference` typedefs
  - No need to implement `construct()` or `destroy()` member functions
  - Only need to specify properties and services that are not the defaults provided by `allocator_traits`

- However, allocator implementors must now consider
  - How to specify copy/move/swap operations for stateful allocators
  - How to represent pointers for non-traditional addressing

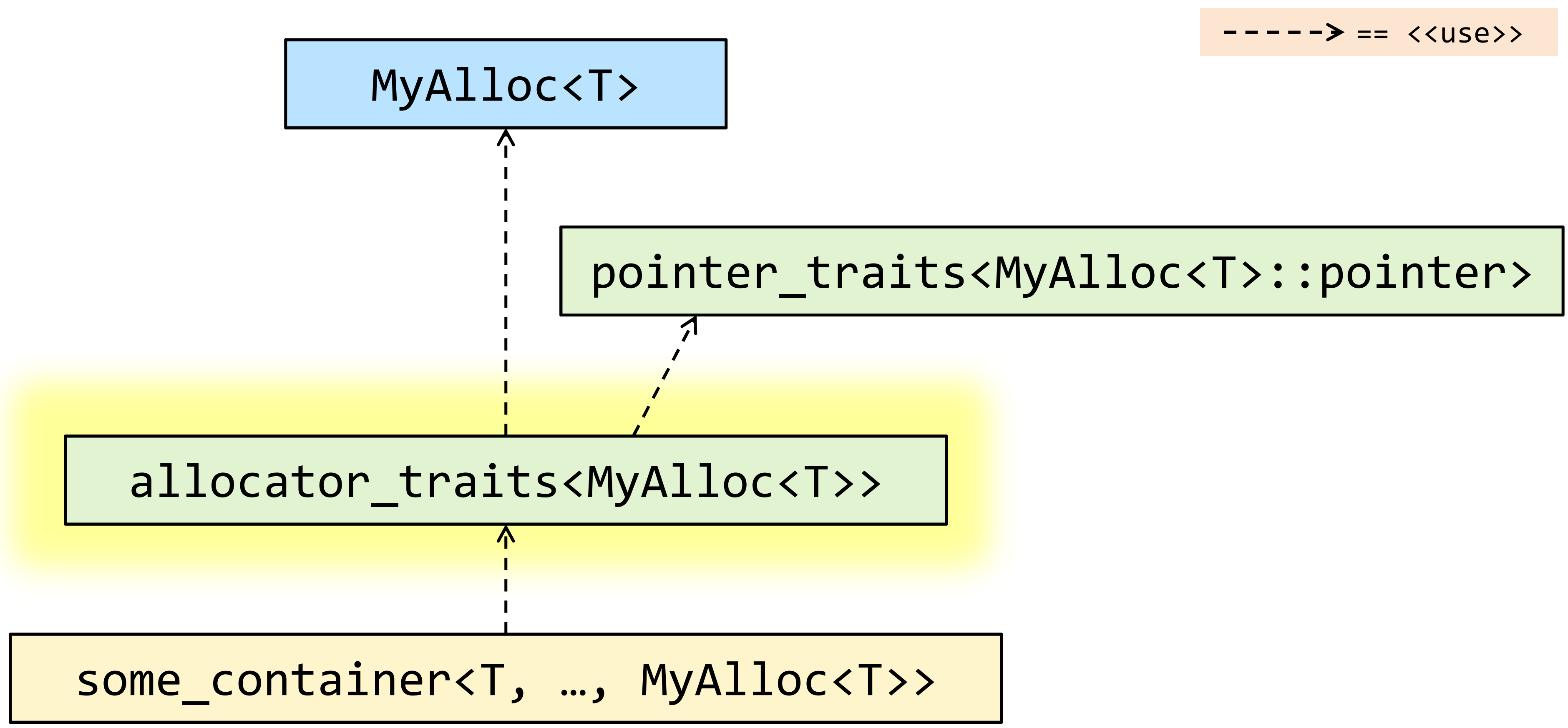
Modern C++ Allocator Model – Containers' Perspective



Example Container – Part 1

```
template<class T, class Alloc = allocator<T>>
class some_container
{
  public:
    using value_type      = T;
    using allocator_type  = Alloc;
    using alloc_traits    = std::allocator_traits<allocator_type>;

    using pointer         = typename alloc_traits::pointer;
    using const_pointer   = typename alloc_traits::const_pointer;
    using size_type       = typename alloc_traits::size_type;
    using difference_type = typename alloc_traits::difference_type;

    using reference       = value_type&;
    using const_reference = value_type const&;
    using iterator        = implementation_defined_stuff;
    using const_iterator  = const_implementation_defined_stuff;
    ...
};
```

Modern C++ Allocator Model – Allocator Traits



Allocator Traits – The Star of the Show

```
std::allocator_traits<Alloc>
```

- Provides a uniform interface to allocators used by the containers
  - Attempts to reflect certain important properties from a target allocator

- Picks sensible defaults for the properties and functions not defined by the target allocator
  - Supplies the boilerplate that was part of C++03 allocators
  - Backwardly-compatible with C++03 allocators

- Allows customization by using non-default properties and functions defined by the target allocator

Allocator Traits Overview

```
template<class Alloc> struct allocator_traits
{
    using allocator_type     = Alloc;
    using value_type         = typename Alloc::value_type;
    using pointer            = INFERRED;
    using const_pointer      = INFERRED;
    using void_pointer       = INFERRED;
    using const_void_pointer = INFERRED;
    using difference_type    = INFERRED;
    using size_type          = INFERRED;

    using propagate_on_container_copy_assignment = INFERRED;    //- POCCA
    using propagate_on_container_move_assignment = INFERRED;    //- POCMA
    using propagate_on_container_swap            = INFERRED;    //- POCS
    using is_always_equal                        = INFERRED;    //- IZEQ

    template <class T> using rebind_alloc  = INFERRED;
    template <class T> using rebind_traits = allocator_traits<rebind_alloc<T>>;
    ...
};
```

Allocator Traits Overview

```
template<class Alloc> struct allocator_traits
{
    ...

    static  pointer   allocate(Alloc& a, size_type n);
    static  pointer   allocate(Alloc& a, size_type n, const_void_pointer hint);
    static  void      deallocate(Alloc& a, pointer p, size_type n);

    template <class T, class... Args>
    static  void      construct(Alloc& a, T* p, Args&&... args);
    template <class T>
    static  void      destroy(Alloc& a, T* p);

    static  size_type max_size(Alloc const& a) noexcept;
    static  Alloc     select_on_container_copy_construction(Alloc const& a);
};
```

Allocator Traits Overview

```
template<class Alloc> struct allocator_traits
{
    ...
    using pointer            = typename Alloc::pointer
                             | value_type*;

    using const_pointer      = Alloc::const_pointer
                             | pointer_traits<pointer>::rebind<const value_type>;

    using void_pointer       = Alloc::void_pointer
                             | pointer_traits<pointer>::rebind<void>;

    using const_void_pointer = Alloc::const_void_pointer
                             | pointer_traits<pointer>::rebind<const void>;

    using difference_type    = Alloc::difference_type
                             | pointer_traits<pointer>::difference_type;

    using size_type          = Alloc::size_type
                             | make_unsigned_t<difference_type>;
    ...
};
```

Allocator Traits Overview

```
template<class Alloc> struct allocator_traits
{
    ...
    using propagate_on_container_copy_assignment
                                        = Alloc::propagate_on_container_copy_assignment
                                        | std::false_type;

    using propagate_on_container_move_assignment
                                        = Alloc::propagate_on_container_move_assignment
                                        | std::false_type;

    using propagate_on_container_swap
                                        = Alloc::propagate_on_container_swap
                                        | std::false_type;

    using is_always_equal               = Alloc::is_always_equal
                                        | std::is_empty<Alloc>::type;

    ...
};
```

Allocator Traits Overview

```
template<class Alloc> struct allocator_traits
{
    ...

    template<class T> using rebind_alloc  = Alloc::rebind<T>::other
                                          | SomeAlloc<T,Args..> if Alloc is a SomeAlloc
                                          | ill-formed;

    template<class T> using rebind_traits = allocator_traits<rebind_alloc<T>>;

    ...
};
```

Allocator Traits Overview

```
template<class Alloc> struct allocator_traits
{

    static  pointer   allocate(Alloc& a, size_type n)
                      {
                          return a.allocate(n);
                      }

    static  pointer   allocate(Alloc& a, size_type n, const_void_pointer hint)
                      {
                          return a.allocate(n, hint);
                               | a.allocate(n);
                      }

    static  void      deallocate(Alloc& a, pointer p, size_type n)
                      {
                          a.deallocate(p, n);
                      }
    ...
};
```

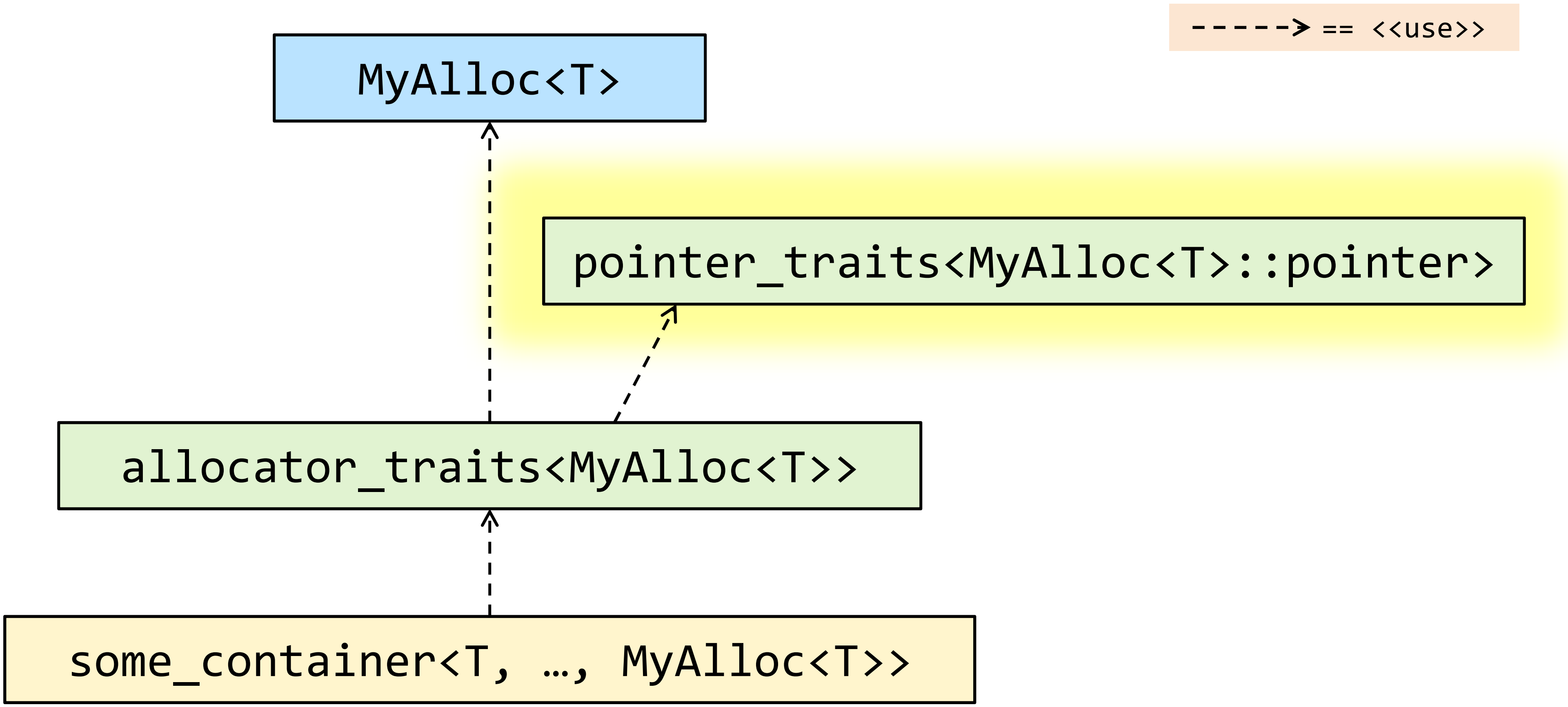
Allocator Traits Overview

```
template<class Alloc> struct allocator_traits
{
    ...
    template <class T, class... Args>
    static  void  construct(Alloc& a, T* p, Args&&... args)
                  {
                      a.construct(p, std::forward<Args>(args)...);
                  |
                      ::new (static_cast<void*>(p)) T(std::forward<Args>(args)...);
                  }

    template <class T>
    static  void  destroy(Alloc& a, T* p);
                  {
                      a.destroy(p);
                  |
                      p->~T();
                  }
    ...
};
```

Allocator Traits Overview

```
template<class Alloc> struct allocator_traits
{

    static  size_type max_size(Alloc const& a) noexcept
                      {
                          return a.max_size();
                               | numeric_limits<size_type>::max()/sizeof(value_type);
                      }

    static  Alloc     select_on_container_copy_construction(Alloc const& a)
                      {
                          return a.select_on_container_copy_construction();
                               | a;
                      }
};
```

Modern C++ Allocator Model – Pointer Traits



Pointer Traits

- Describes the properties of pointers and pointer-like types
- Provides the type of pointer differences
- Provide the pointed-to type
  - Given the type `T*`, get the type `T`
  - Given the type `fancy_ptr<T>`, get the type `T`
- Provide a transformation from one pointer type to another pointer type
  - Given the type `T*`, get the type `U*`
  - Given the type `fancy_ptr<T>`, get the type `fancy_ptr<U>`

Pointer-Like Types (AKA Fancy / Synthetic Pointers)

- Mentioned 4 times in the standard, but the only substance is in the requirements for *NullablePointer* (see Table 28 in N4687):
  - Must satisfy several requirements:
    - *EqualityComparable*, *DefaultConstructible*, *CopyConstructible*, *CopyAssignable*, and *Destructible*
  - Have swappable lvalues
  - Default initialization my produce an indeterminate result; using may lead to UB
  - Value initialization produces a null result
  - Construction with / assignment from `nullptr` produces a null result
  - May be contextually converted to `bool`
  - Certain fundamental operations (see Table 28) may not throw exceptions

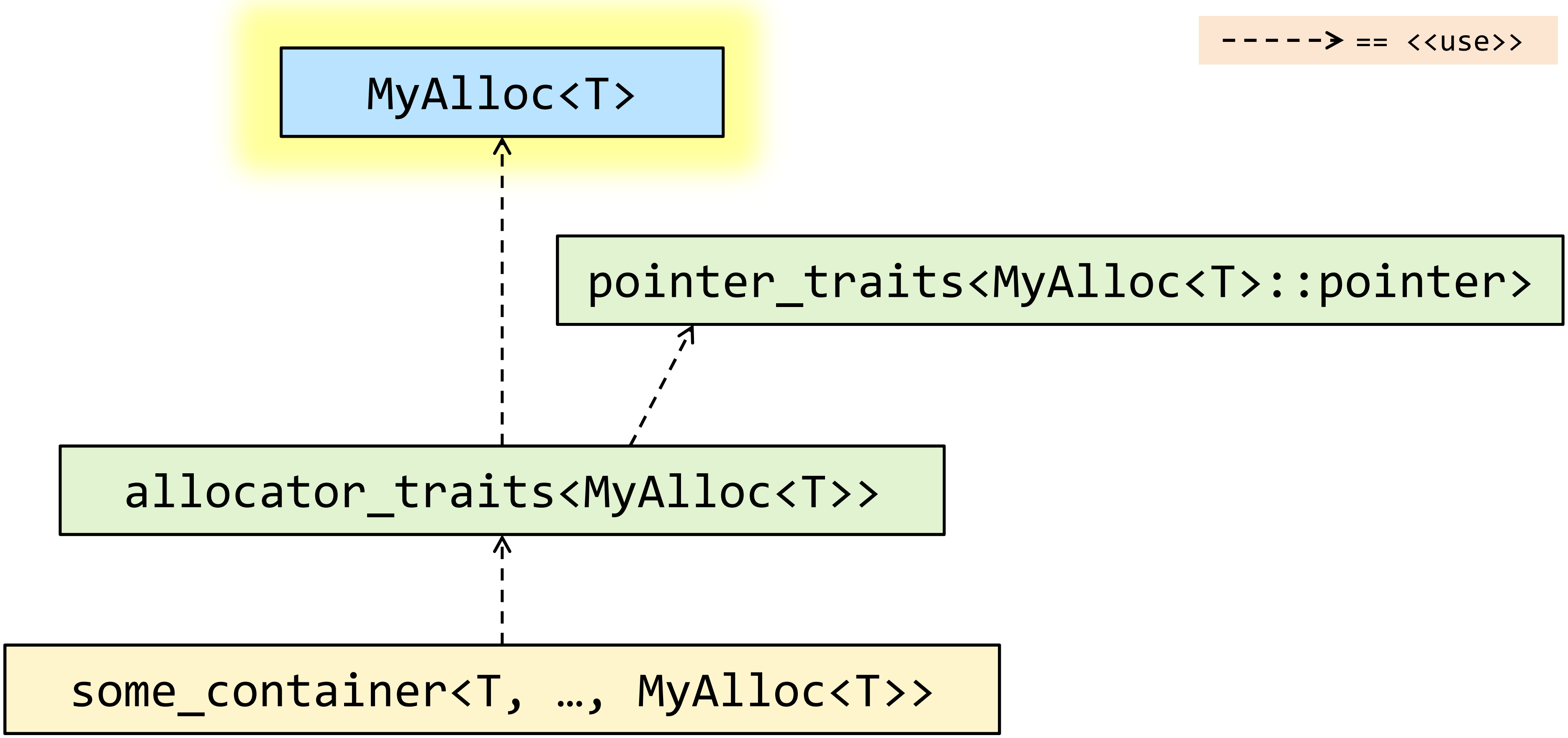
Pointer Traits

```
template<class T> struct pointer_traits;

template<class T>
struct pointer_traits<T*>
{
    using pointer         = T*;
    using element_type    = T;
    using difference_type = ptrdiff_t;

    template<class U> using rebind = U*;

    static  pointer pointer_to(element_type& r) noexcept { return std::addressof(r); }
};
```

Pointer Traits

```
template<class Ptr>
struct pointer_traits<Ptr>
{
    using pointer         = Ptr;

    using element_type    = typename Ptr::element_type
                          | T if Ptr is a SomePointer<T,Args...>
                          | ill-formed;

    using difference_type = typename Ptr::difference_type
                          | ptrdiff_t;

    template<class U>
          using rebind = Ptr::template rebind<U>
                          | SomePointer<U,Args...> if Ptr is a SomePointer<T,Args...>;
                          | ill-formed

    static  pointer pointer_to(element_type& r) { return Ptr::pointer_to(r); }
};
```

Modern C++ Allocator Model – The Allocator



A Minimal Allocator

```
template<class T>
struct minimal_allocator
{
    using value_type = T;

    minimal_allocator(some_params) {}
    template<class U> minimal_allocator(minimal_allocator<U> const&) noexcept {}

    T*    allocate(size_t n);
    void  deallocate(T* p, size_t);
};

template<class T> bool
operator ==(minimal_allocator<T> const&, minimal_allocator<T> const&);

template<class T> bool
operator !=(minimal_allocator<T> const&, minimal_allocator<T> const&);
```

The Modern Default Allocator

```
template<class T>  class allocator
{
  public:
    using value_type                             = T;
    using propagate_on_container_move_assignment = true_type;
    using is_always_equal                        = true_type;

    allocator() noexcept;
    allocator(const allocator&) noexcept;
    template<class U> allocator(const allocator<U>&) noexcept;
    ~allocator();

    T*    allocate(size_t n);
    void  deallocate(T* p, size_t n);
};

template<class T, class U>
bool operator ==(allocator<T> const&, allocator<U> const&) noexcept;

template<class T, class U>
bool operator !=(allocator<T> const&, allocator<U> const&) noexcept;
```

The Modern Default Allocator – Allocate / Deallocate

```cpp
template<class T> inline T*
allocator<T>::allocate(size_t n)
{
    return static_cast<T*>(::operator new(n * sizeof(T)));
}


template<class T> inline void
allocator<T>::deallocate(T* p, size_t)
{
    ::operator delete(p);
}
```

The Modern Default Allocator – Comparison

```
template<class T, class U> inline bool
operator ==(allocator<T> const&, allocator<U> const&) noexcept
{
    return true;
}


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

There’s a New Kid in Town

- Polymorphic memory resources (PMR)
  - Defined in namespace `std::pmr`
  - Provide runtime polymorphism with single type argument to containers
  - Client allocators store a pointer to a base class memory resource
  - No lateral propagation – an allocator sticks for life

Polymorphic Memory Resource (PMR)

```
class pmr::memory_resource
{
    static constexpr size_t max_align = alignof(max_align_t);

  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;

  private:
    virtual void*   do_allocate(size_t bytes, size_t alignment) = 0;
    virtual void    do_deallocate(void* p, size_t bytes, size_t alignment) = 0;
    virtual bool    do_is_equal(const memory_resource& other) const noexcept = 0;
};
```

Polymorphic Memory Resource (PMR)

```
class monotonic_buffer_resource : public memory_resource
{
  public:
    monotonic_buffer_resource();
    explicit monotonic_buffer_resource(memory_resource *upstream);
    ...
    monotonic_buffer_resource(const monotonic_buffer_resource&) = delete;
    virtual ~monotonic_buffer_resource();

    monotonic_buffer_resource&  operator=(const monotonic_buffer_resource&) = delete;

    void                 release();
    memory_resource*     upstream_resource() const;

  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 memory_resource& other) const noexcept override;
};
```

Polymorphic Allocator

```
template<class T> class polymorphic_allocator
{
    memory_resource*    memory_rsrc;

  public:
    using value_type = T;

    polymorphic_allocator() noexcept;
    polymorphic_allocator(memory_resource* r);
    polymorphic_allocator(const polymorphic_allocator& other) = default;
    template <class U>
    polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;

    polymorphic_allocator&  operator=(const polymorphic_allocator& rhs) = delete;

    T*      allocate(size_t n);
    void    deallocate(T* p, size_t n);

    polymorphic_allocator   select_on_container_copy_construction() const;
    ...
};
```

PMR Type Aliases

```
namespace pmr    //- A partial list of type aliases now provided in std::pmr
{
    template <class charT, class traits = char_traits<charT>>
    using basic_string = std::basic_string<charT, traits, polymorphic_allocator<charT>>;

    using string = basic_string<char>;
    ...

    template <class T>
    using deque = std::deque<T, polymorphic_allocator<T>>;
    ...

    template <class Key, class Compare = less<Key>>
    using set = std::set<Key, Compare, polymorphic_allocator<Key>>;
    ...

    template <class Key, class Hash = hash<Key>, class Pred = equal_to<Key>>
    using unordered_set = std::unordered_set<Key, Hash, Pred,
                                                   polymorphic_allocator<Key>>;
}
```

A Container’s Point of View

Example Container – Part 2

```
template<class T, class Alloc = allocator<T>>
class some_container
{
  public:
    using value_type      = T;
    using allocator_type  = Alloc;
    using alloc_traits    = std::allocator_traits<allocator_type>;

    using pointer         = typename alloc_traits::pointer;
    using const_pointer   = typename alloc_traits::const_pointer;
    using size_type       = typename alloc_traits::size_type;
    using difference_type = typename alloc_traits::difference_type;

    using reference       = value_type&;
    using const_reference = value_type const&;
    using iterator        = implementation_defined_stuff;
    using const_iterator  = const_implementation_defined_stuff;
    ...
};
```

Example Container – Part 2

```
template<class T, class Alloc<T>>
class some_container
{
  public:
    ...
    some_container(some_container const& other);
    some_container(some_container const& other, allocator_type const& alloc);

    some_container(some_container&& other);
    some_container(some_container&& other, allocator_type const& alloc);

    some_container&  operator =(some_container const& other);
    some_container&  operator =(some_container&& other);

    swap(some_container& other);

    ...
};
```

Example Container – Part 2

```
template<class T, class Alloc<T>>
class some_container
{
    ...

  private:
    impl_data       m_impl;
    allocator_type  m_alloc;

    template<class Iter>
    void    assign_from(Iter f, Iter l);
    void    clear_and_deallocate_memory()
};
```

Example Container – Copy Construction

```
template<class T, class Alloc>
some_container<T,Alloc>::some_container(some_container const& other)
:   m_impl()
,   m_alloc(traits_type::select_on_container_copy_construction(other.m_alloc))
{
     this->assign_from(other.cbegin(), other.cend());
}


template<class T, class Alloc>
some_container<T,Alloc>::some_container(some_container const& other,
                                        allocator_type const& alloc)
:   m_impl()
,   m_alloc(alloc)
{
    this->assign_from(other.cbegin(), other.cend());
}
```

Example Container – Move Construction

```
template<class T, class Alloc>
some_container<T,Alloc>::some_container(some_container&& other)
:   m_impl(std::move(other.m_impl))
,   m_alloc(std::move(other.m_alloc))
{}


template<class T, class Alloc>
some_container<T,Alloc>::some_container(some_container&&      other,
                                        allocator_type const& alloc)
:   m_impl(std::move(other.m_impl))
,   m_alloc(alloc)
{}
```

Example Container – Copy Assignment

```cpp
template<class T, class Alloc>  some_container&
some_container<T,Alloc>::operator =(some_container const& other)
{
    if (&other != this)
    {
        if (alloc_traits::POCCA == std::true_type)
        {
            if (this->m_alloc != other.m_alloc)
            {
                this->clear_and_deallocate_memory();
            }
            this->m_alloc = other.m_alloc;
        }
        this->assign_from(other.cbegin(), other.cend());
    }
    return *this;
}
```

Example Container – Move Assignment

```
template<class T, class Alloc>  some_container&
some_container<T,Alloc>::operator =(some_container&& rhs)
{
    if (alloc_traits::POCMA == std::true_type)
    {
        this->clear_and_deallocate_memory();
        this->m_alloc = std::move(rhs.m_alloc);
        this->m_impl  = std::move(rhs.m_impl);
    }
    else if (alloc_traits::is_always_equal == std::true_type ||
             this->m_alloc == rhs.m_alloc)
    {
        this->clear_and_deallocate_memory();
        this->m_impl = std::move(rhs.m_impl);
    }
    else
    {
        this->assign(std::move_iterator(rhs.begin()), std::move_iterator(rhs.end()));
    }
    return *this;
}
```

Example Container – Swap

```
template<class T, class Alloc>  void
some_container<T,Alloc>::swap(some_container& other)
{
    if (&other != this)
    {
        if (alloc_traits::POCS == std::true_type)
        {
            std::swap(this->m_impl, other.m_impl);
            std::swap(this->m_alloc, other.m_alloc);
        }
        else if (alloc_traits::is_always_equal == std::true_type ||
                 this->m_alloc == rhs.m_alloc)
        {
            std::swap(this->m_impl, other.m_impl);
        }
        else    //- POCS is std::false_type and this->m_alloc != other.m_alloc
        {
                //- This is undefined behavior.
        }
    }
}
```

A Few Guidelines

Thinking About Memory Allocation – Design Decisions

- Plumbing
  - Public Interface: Traditional or PMR
- Structural Management
  - Addressing Model
  - Storage Model
  - Pointer Interface
  - Allocation Strategy
- Concurrency Management
  - Thread Safety
  - Transaction Safety

Thinking About Your New Allocator – Traditional

- Review the minimal and default allocator interfaces
- Decide how to specify properties and functionality
  - Everything in the allocator type; or,
  - Partially specializing `std` traits types
- Synthetic pointers or ordinary pointers?
  - If synthetic, don't forget your pointer type's re-binder
  - If synthetic, you must define a nested `to_pointer()` member function
- Do specify your allocator's nested re-binder
- Do specify your allocator's nested `value_type` alias

Thinking About Your New Allocator – Traditional

- If instances always compare equal?
  - Define a nested type alias `is_always_equal` set to `true_type`
- If instances *do not* propagate on copy construction
  - Define `select_on_container_copy_construction()` that returns a temporary value of your allocator type
- If instances propagate on copy assignment,
  - Define nested alias `propagate_on_container_copy_assignment` as `true_type`
- If instances propagate on move assignment,
  - Define nested alias `propagate_on_container_move_assignment` as `true_type`
- If instances propagate on swap,
  - Define nested alias `propagate_on_container_swap` as `true_type`

Thinking About Your New Allocator - PMR

- Which `pmr` base class?
  - **`pmr::memory_resource`**
  - There are others, derived from `pmr::memory_resource` …
    - `synchronized_pool_resource`
    - `unsynchronized_pool_resource`
    - `monotonic_buffer_resource`

Thinking About Your New Allocator

- If you don't need fancy pointers, and you can live with a little extra overhead in each container
  - Create a new PMR type, derived from `pmr::memory_resource`

- Otherwise, you'll need to jump into the deep end of the traditional allocator pool

Thank You for Attending!

https://gitlab.com/BobSteagall/talks/CppCon2017 (that's Git<u>Lab</u>)