

Doc No: N2840=09-0030

Date: 2009-03-05

Author: Pablo Halpern
Cilk Arts, Inc.

phalpern@halpernwrightsoftware.com

Defects and Proposed Resolutions for Allocator Concepts (Rev 2)

Contents

National Body Comments Addressed in this Paper	1
Summary	2
Changes from N2829 (Rev 1).....	2
Changes from N2810	2
Document Conventions	2
Typographical and Editorial Errors	3
Allocator concept does not match all C++03 allocators	3
Allocator::rebind is different from C++03 rebind.....	4
Construct Method is Limited to value_type.....	4
construct_element Function is Unnecessary.....	5
Allocator Propagation Relies on Traits Instead of Concepts	6
is_scoped_allocator Trait is not Used.....	6
scoped_allocator_adaptor has errors	7
Two Types of scoped_allocator_adaptors.....	7
Proposed Wording.....	8
Typographical and Editorial Corrections.....	8
Revised Allocator Concept	9
Legacy Allocators.....	10
Rename rebind to rebind_type.....	14
Replace construct_element with construct	15
Allocator Propagation	16
Remove is_scoped_allocator trait.....	19
Modified scoped_allocator_adaptor	19
References	26

National Body Comments Addressed in this Paper

The previous version of this paper ([N2810](#)) comprises the proposed resolution for CD comment numbered US 75. The issues with pair have been split off into a separate paper:

[N2834](#). This proposal provides formal wording for the resolutions to UK 223 and UK 224 and addresses most of the concerns of US 65 and US 74.

Summary

The addition of concepts for allocators in the standard library is incomplete and has a number of defects. Each defect is listed below with a proposed resolution. For easier reference, there is no formal wording for individual proposed resolutions. Instead, formal wording for all of the proposed resolutions is merged into one set of changes at the end of the document.

Changes from N2829 (Rev 1)

- Updated references to refer to October 2009 CD instead of the WP.
- Clarified some text and fixed numerous typos and small omissions.
- Simplified `AllocatableElement` and removed the `has_custom_construct` trait.
- More thorough cleaning of remnants of the pre-concept allocator stuff in response to NB comments.

Changes from N2810

- Added more specific proposed wording.
- Moved discussion for reducing the number of `pair` constructors into a separate paper ([N2834](#)).
- Removed issue about `vector` and `string` being under-constrained. `Vector` and `basic_string` will work fine with non-native pointer types.
- Replaced the `AdvancedAllocator` concept with a `custom_construct` associated type in the `Allocator` concept.
- This version of the proposal uses `rebind_type` instead of `related_instance` as the replacement for `rebind` in the `Allocator` concept.

Document Conventions

All section names and numbers are relative to the October 2008 CD, N2800.

Existing and proposed working paper text is indented and shown in dark blue. Small edits to the working paper are shown with ~~red strikeouts for deleted text~~ and green underlining for inserted text within the indented

blue original text. Large proposed insertions into the working paper are shown in the same dark blue indented format (no green underline).

Comments and rationale mixed in with the proposed wording appears as shaded text.

Requests for LWG opinions and guidance appear with light (yellow) shading. It is expected that changes resulting from such guidance will be minor and will not delay acceptance of this proposal in the same meeting at which it is presented.

Typographical and Editorial Errors

Description of Issue

The latest WP contains a number of errors of a typographical or editorial nature.

Proposed Resolution

Correct the errors as described in the proposed wording section of this paper.

Allocator concept does not match all C++03 allocators

Description of Issue

The reason that `Allocator` is declared `auto` is to support backwards compatibility with C++03 allocators. Otherwise, the `Allocator` concept is not the kind of concept that would normally be declared `auto`. However, declaring it `auto` does not fully succeed at providing this backwards compatibility. C++03 allocators do not have a `generic_pointer` type nor a `variadic_construct` function. Moreover, the non-variadic `construct` function in the C++03 allocator takes a `pointer` argument instead of a `T*` argument as in the concept. As a result, there are some C++03 allocators for which there would be no automatically-generated concept map.

Proposed Resolution

Remove the `auto` modifier from the `Allocator` concept. This change will make `Allocator` cleaner (by avoiding gratuitous use of `auto`) and will allow additional evolution without the constraint of automatic compatibility with C++03 allocators. To address the compatibility problem, add a `LegacyAllocator` `auto` concept and `concept_map` for `Allocator` to automatically adapt any class meeting the requirements of a C++03.

Allocator::rebind is different from C++03 rebind

Description of Issue

The `rebind` template in the `Allocator` concept serves the same purpose as the `rebind` template in the C++03 allocator requirements, but uses a different syntax for both definition and declaration. In the case of a C++03 allocator, rebinding is done by referencing `Alloc::rebind<U>::other`, whereas in C++0x, rebinding is done by referencing: `Allocator<Alloc>::rebind<U>` (no other nested type). Moreover, the declaration of `rebind` in a C++0x concept map is simpler than in a C++03 allocator: `template <class T> rebind = MyAlloc<T>;` instead of `template <class T> struct rebind { typedef MyAlloc<T> other; };`. These differences are confusing and could cause strange compilation errors when adding constraints to an unconstrained container template.

Proposed Resolution

Rename the `rebind` template in the `Allocator` concept to something else. Some ideas are `rebind_type`, `retype`, `related_type`, `sibling`, or `sibling_allocator`. I used `rebind_type` in the proposed wording because it will be familiar to C++03 programmers, yet different enough for the compiler to tell you when you've used the wrong one. The definition of `rebind_type` in the concept map for legacy allocators can still be implemented in terms of `rebind`.

Construct Method is Limited to `value_type`

Description of Issue

The `construct` method in the `Allocator` concept only constructs objects of type `value_type`. This constraint can lead to inconvenient and sometimes inefficient uses of `rebind_type` in order to construct objects of different types. For example, a container type might allocate objects of type `Node<T>` from an allocator, `alloc_`, of type `allocator_type` `== Alloc<Node<T>>`. However, some parts of the `Node` might be initialized independently of the `T` object contained within it. Initializing an inner object of type `U` would require the use of `rebind_type` as follows:

```
Allocator<allocator_type>::rebind_type<U>(alloc_).construct(p, args);
```

The above `construct` is not only hard to read, but it constructs a temporary object of type `Alloc<T>` just to call its `construct` method.

Proposed Resolution

(Note: This resolution is refined by the resolution of the next issue.) We have already changed `construct` to a template in order to support `emplace`. It is a small matter, then, to templatize the pointer argument as well as the constructor arguments:

```
template<typename T, typename... Args>
    requires HasConstructor<T, Args&&...>
    void X::construct(T* p, Args&&... args)
{
    ::new ((void*) p) T(forward<Args>(args)...);
}
```

`construct_element` Function is Unnecessary

Description of Issue

The `construct_element` function was originally introduced to centralize the moderately-complex, traits-based dispatch of the construction of elements based a number of allocator and element traits. The advent of concepts allows us to eliminate the `construct_element` meta-programming layer and express constraints directly in the declaration of the `construct` member for each allocator.

Proposed Resolution

Remove the global `construct_element` template and remove the `construct_element` function from the `AllocatableElement` concept. Move the `construct` function from the `Allocator` concept to the `AllocatableElement` concept, as shown in the proposed wording section of this paper. This change will also lift the restriction on `construct` described in the previous issue. Change other uses of `construct_element` to use `construct` directly. For consistency, move `destroy` from `Allocator` to `AllocatableElement`, as well.

This change, however, will remove the default implementation of `construct` because such a default implementation would potentially bypass constraints required by a specialized allocator (such as a scoped allocator). However, the typical implementation of `construct` can be inserted into an allocator with a simple cut-and-paste of a tiny amount of boilerplate code (with no changes to the boilerplate), so losing the default implementation will not make much difference to an allocator author.

Allocator Propagation Relies on Traits Instead of Concepts

Description of Issue

The allocator propagation traits, `allocator_propagate_never`, `allocator_propagate_on_copy_construction`, `allocator_propagate_on_move_assignment`, and `allocator_propagate_on_copy_assignment` are pre-concept ways to express what can now be expressed more cleanly with concepts. These traits control the behavior of allocator propagation for a set of static member functions in the `allocator_propagation_map` structure. The system is unnecessarily complex now that we have an `Allocator` concept into which we can directly insert the propagation functions with default implementations.

Proposed Resolution

Add the following five functions and default implementations to the `Allocator` concept. (The fifth function is for move-construction, which is not separately accounted-for in the current WP):

```
Alloc select_on_container_copy_construction(const Alloc& x) { return x; }
Alloc select_on_container_move_construction(Alloc&& x) { return move(x); }
void do_on_container_copy_assignment(Alloc& to, Alloc& from) { }
void do_on_container_move_assign(Alloc& to, Alloc&& from) { }
void do_on_container_swap(Alloc& a, Alloc& b) { }
```

Refinements of `Allocator` for individual allocator types can override these defaults as desired.

See the proposed wording section for an embodiment of this proposed resolution.

`is_scoped_allocator` Trait is not Used

Description of Issue

The `is_scoped_allocator` trait is still in the WP, but is not referenced since allocator concepts were introduced. A concept-based approach needs to be introduced to replace the purpose of the `is_scoped_allocator` trait – i.e., to dispatch the element-construction functionality based on allocator type.

Proposed Resolution

Remove the `is_scoped_allocator` trait. Constrain the `construct` member function of any scoped allocator such that an element must be `ConstructibleWithAllocator` using the inner allocator type. Combined with the proposed resolution for the previous issue

(moving the `construct` function into `AllocatableElement`), this resolution removes scoped allocator notions from unrelated parts of the standard, reducing clutter and confusion.

scoped_allocator_adaptor has errors

Description of Issue

The `scoped_allocator_adaptor` templates have errors in them, some of which cause them not to model the `Allocator` concept in every detail. Specifically:

- There are places where `void` is used instead of *unspecified allocator type*.
- The `construct` and `destroy` methods take `pointer` instead of `value_type*`. (But if the resolution to the `construct` issue is accepted, it should take a pointer to template-argument type.)
- Some `Allocator` constraints are missing.

Proposed Resolution

Correct the errors. See related issues in this paper for other changes that may apply.

Two Types of scoped_allocator_adaptors

Description of Issue

The WP describes `scoped_allocator_adaptor` as a class template with two template parameters, one for the outer allocator type and one for the inner allocator type. A specialization of `scoped_allocator_adaptor` takes only one template parameter. In the latter case, not only are both the outer and inner allocators the same type, they are also the same object. Thus there is a difference between `scoped_allocator_adaptor<A1, A1>`, which holds two distinct instances of type `A1`, and `scoped_allocator_adaptor<A1>`, which holds a single instance of type `A1`. This use of a default parameter has already caused significant confusion.

Proposed Resolution

Use separate names, `scoped_allocator_adaptor` and `scoped_allocator_adaptor2` for the single-parameter and dual-parameter adaptor templates, respectively. Remove the default argument of *unspecified allocator type*.

Proposed Wording

Typographical and Editorial Corrections

Section 20.7 [memory], Header <memory> synopsis, add missing close-angle-bracket and remove unnecessary semicolon:

```
// 20.7.6, the default allocator:
template <class T> class allocator;
template <ObjectType T>
    concept_map Allocator<allocator<T> > { };
```

Section 20.7 [memory], change all occurrences of `pointer::reference` to `pointer::result_type`:

```
requires Convertible<pointer, const_pointer>
    && Convertible<pointer, generic_pointer>
    && SameType<pointer::reference, result_type, value_type&>
    && SameType<pointer::reference, result_type, reference>;
```

(and a number of other occurrences).

Section 21.3, remove reference to obsolete traits:

```
template <class charT, class traits, class Alloc
struct constructible_with_allocator_suffix<
basic_string<charT, traits, Alloc> > : true_type { };
```

Section 23.1, Table 82, remove rows dealing with traits that have been obsolete by concepts:

uses_allocator<X, Q>	derived from true_type or false_type	true_type if Q is convertible to A	compile time
constructible_with_allocator_suffix<X>	derived from true_type		Compile time

Section 30.5.1 [futures.overview], synopsis, correct use of obsolete traits:

```
template <class R, class Alloc>
struct uses_allocator<promise<R>, Alloc>;
concept map UsesAllocator<promise<R>, Alloc> {
typedef Alloc allocator_type;
}
template <class R>
struct constructible_with_allocator_prefix<promise<R> >;
```

Remove section 30.5.7 [futures.allocators] entirely:

~~30.5.7 Allocator Templates [futures.allocators]~~

~~template <class R, Etc...>~~

Revised Allocator Concept

In section 20.7 [memory], remove the `auto` keyword from the Allocator concept, correct the spelling of `HasDereference` and `rebind_type`:

// 20.7.2.2 Allocator concepts

```
auto concept Allocator<#typename AlleeX> see below;  
auto concept LegacyAllocator<typename X> see below  
template <LegacyAllocator X> concept map Allocator<X> see below
```

In section 20.7.2.2 [allocator.concepts], modify the Allocator concept and add a concept map as follows:

```
auto concept Allocator<typename X> :  
    CopyConstructible<X>, EqualityComparable<X> {  
  
    ObjectType value_type = typename X::value_type;  
    requires FreeStoreAllocatable<value_type>;  
  
    DereferenceableHasDereference pointer = see below;  
    DereferenceableHasDereference const_pointer = see below;  
    requires Regular<pointer>  
        && RandomAccessIterator<pointer>  
        && Regular<const_pointer>  
        && RandomAccessIterator<const_pointer>;  
    SignedIntegralLike difference_type =  
        RandomAccessIterator<pointer>::difference_type;  
    typename generic_pointer = void*;  
    typename const_generic_pointer = const void*;  
    typename reference = value_type&;  
    typename const_reference = const value_type&;  
    UnsignedIntegralLike size_type = see below;  
    template<ObjectType T> class rebind type == see below;  
  
    requires Destructible<value_type>;  
    requires Convertible<pointer, const_pointer>  
        && Convertible<pointer, generic_pointer>  
        && SameType<pointer::result_type, value_type&>  
        && SameType<pointer::result_type, reference>;  
    requires Convertible<const_pointer, const_generic_pointer>  
        && SameType<const_pointer::result_type, const value_type&>  
        && SameType<const_pointer::result_type, const_reference>;  
    requires SameType<rebind type<value_type>, X>;  
    requires SameType<generic_pointer  
        , rebind type<unspecified unique type>::generic_pointer>;  
        // see description of generic_pointer, below  
    requires SameType<const_generic_pointer  
        , rebind type< unspecified unique type>::const_generic_pointer>;  
        // see description of generic_pointer, below  
  
    pointer X::allocate(size_type n);
```

```

pointer X::allocate(size_type n, const_generic_pointer p);
void X::deallocate(pointer p, size_type n);
size_type X::max_size() const {
    return numeric_limits<size_type>::max(); }

template<ObjectType T> X::X(const rebind_type<T>& y);

template<typename... Args>
requires HasConstructor<value_type, Args&&...>
void X::construct(value_type* p, Args&&... args)
{
    ::new ((void*) p) value_type(forward<Args>(args)...);
}

void X::destroy(value_type* p) {
    addressof(*p) -> value_type();
}

pointer X::address(reference r) const {
    return addressof(r); // see below
}

const_pointer X::address(const_reference r) const {
    return addressof(r); // see below
}

X select on container copy construction(const X& x) { return x; }
X select on container move construction(X&& x) { return move(x); }
void do on container copy assignment(X& to, const X& from) { }
void do on container move assignment(X& to, X&& from) { }
void do on container swap(X& a, X& b) { }
}

```

Remove the definition of `construct` at paragraph 15:

```

template<typename... Args>
requires HasConstructor<value_type, Args&&...>
void X::construct(value_type* p, Args&&... args);

```

~~Effects: Calls the constructor for the object at p, using the args constructor arguments.~~

~~Default: ::new ((void*) p) value_type(forward<Args>(args)...);~~

Legacy Allocators

At the start of 20.7.2.2 [allocator.concepts], after the definition of the `Allocator` concept, add a new sub-section:

20.7.2.2.1 Support for legacy allocators [allocator.concepts.legacy]

Classes that meet the allocator requirements described in table 32 of [ISO/IEC 14882:2003](#) are known as *legacy Allocators*. The `LegacyAllocator` auto concept abstracts the requirements of legacy allocators. A concept map adapts `LegacyAllocator` to the `Allocator` concept. [*Note: not all legacy allocator requirements can be precisely described using concepts. In particular, `rebind` is under-constrained and the second argument of `allocate(p, u)` is discarded in the default implementation to make up for the fact that it is over-constrained. Legacy allocators that are not a precise fit for the `LegacyAllocator` concept can be used as allocators by supplying a concept map adapting them to the `Allocator` concept. – end note]*

```
auto concept LegacyAllocator<typename X> :
    DefaultConstructible<X>, CopyConstructible<X>, EqualityComparable<X> {

    HasDereference pointer = X::pointer;
    HasDereference const_pointer = X::const_pointer;
    typename generic_pointer = void*;
    typename const_generic_pointer = const void*;
    typename reference = X::reference;
    typename const_reference = X::const_reference;
    ObjectType value_type = typename X::value_type;
    UnsignedIntegralLike size_type = X::size_type;
    SignedIntegralLike difference_type = X::difference_type;
    template<ObjectType T> struct rebind = see below;

    requires Destructible<value_type>;
    requires Regular<pointer>
        && RandomAccessIterator<pointer>
        && Regular<const_pointer>
        && RandomAccessIterator<const_pointer>;
    requires Convertible<pointer, const_pointer>
        && Convertible<pointer, generic_pointer>
        && SameType<pointer::result_type, value_type&>
        && SameType<pointer::result_type, reference>;
    requires Convertible<const_pointer, const_generic_pointer>
        && SameType<const_pointer::result_type, const value_type&>
        && SameType<const_pointer::result_type, const_reference>;
    requires IntegralType<size_type> && IntegralType<difference_type>;

    pointer X::address(reference r) const;
    const_pointer X::address(const_reference r) const;

    pointer X::allocate(size_type n);
    pointer X::allocate(size_type n, generic_pointer p)
        { return X::allocate(n); }

    void X::deallocate(pointer p, size_type n);

    size_type X::max_size() const;

    template<ObjectType T>
        requires see below
        X::X(const typename rebind<T>::other& y);
```

```

// Not used, but part of Table 32
requires CopyConstructible<value_type>
    void X::construct(pointer_type p, const value_type&);

    void X::destroy(pointer p);
}
template <LegacyAllocator X>
concept_map Allocator<X> {
    typedef X::value_type          value_type;
    typedef X::pointer             pointer;
    typedef X::const_pointer       const_pointer;
    typedef X::generic_pointer     generic_pointer;
    typedef X::const_generic_pointer const_generic_pointer;
    typedef X::difference_type     difference_type;
    typedef X::size_type           size_type;
    typedef X::reference            reference;
    typedef X::const_reference     const_reference;

    template<ObjectType T> using rebind_type = see below;
}

```

The rebind<> requirements and the requirements on the rebind-conversion constructor can be implemented using a `__HasOther` and an `__AllocRebindTypeOf` concept:

```

auto concept __HasOther<typename T> {
    typename other = typename T::other;
}

concept __AllocRebindTypeOf<typename Y, typename X> { }

auto concept LegacyAllocator<typename X> : ... {
    ...
    template<ObjectType T> struct rebind;
    requires __Ho = __HasOther<rebind<value_type>>;
    requires SameType<__Ho::other, X>;
    requires __AllocRebindTypeOf<__Ho::other, X>;
    ...
    template<typename Y>
    requires __AllocRebindTypeOf<Y, X>
    X::X(const Y& y);
}

template <LegacyAllocator X, ObjectType T>
requires __Ho = __HasOther<X::rebind<T>>
    concept_map __AllocRebindTypeOf<__Ho::other, X> {}

```

Note that the `construct()` function in the legacy allocator is ignored in the concept-map mapping. [ISO/IEC 14882:2003](https://www.iso.org/standard/62653.html) did not require that `construct` be called by containers and many implementations handled it inconsistently. The addition of the variadic `construct` function makes a direct mapping from C++03 allocators to C++0x allocators impossible. An

adaptor function could construct a temporary object of type `value_type`, then call the C++03 `construct` function, but that would add a new `CopyConstructible` requirement and a quiet inefficiency (caused by an extra copy) for all C++03 allocators even though very few require a custom `construct` function. It seems safer to simply ignore the legacy `construct` function and let the user create their own concept map if necessary.

There is no way to derive a `generic_pointer` type from `pointer`. The default of `void*` will work for most, but not all, legacy allocators. Legacy allocators that use non-raw pointer types, will need an explicit concept map for either `Allocator` or `LegacyAllocator`.

Also, insert a new header immediately following the concept map for `Allocator<LegacyAllocator>` and immediately before the description of `Allocator` members:

[20.7.2.2.2 Allocator and LegacyAllocator members \[allocator.concepts.members\]](#)

Modify the description of `generic_pointer` as follows:

```
typename generic_pointer;  
typename const_generic_pointer;
```

A type that can store value of a pointer (`const_pointer`) from any allocator in the same family as `X` and which will produce the same value when explicitly converted back to that pointer type. For any two allocators `X`, and `Y` of the same family, the implementation of a library facility using `Allocator<X>` and `Allocator<Y>`, is permitted to add additional requirements, `SameType<Allocator<X>::generic_pointer, Allocator<Y>::generic_pointer>` and `SameType<Allocator<X>::const_generic_pointer, Allocator<Y>::const_generic_pointer>` [*Example*:

```
template<ObjectType T, Allocator Alloc = allocator<T> >  
    requires Destructible<T> &&  
    SameType<Alloc::generic_pointer,  
        Alloc::Rebind<list_node<T>>::generic_pointer> &&  
    SameType<Alloc::const_generic_pointer,  
        Alloc::Rebind<list_node<T>>::const_generic_pointer>  
    class list;
```

end example]

Default types: `X::generic_pointer` and `X::const_generic_pointer` if such types exists and `void*` and `const void*` otherwise.

And modify the definition of `rebind` as follows:

```
template<ObjectType T> class rebind_type;
```

Class Template: The associated template `rebind_type` is a template that produces allocators in the same family as `X`: if the name `X` is bound to `SomeAllocator<value_type>`, then `rebind_type<U>` is the same type as `SomeAllocator<U>`. The resulting type `SomeAllocator<U>` shall meet

the requirements of the Allocator concept. ~~The default value for rebind is a template R for which R<U> is X::template rebind<U>::other.~~

```
template<ObjectType T> class rebind;
```

Class Template: The associated template rebind (for LegacyAllocator only) is a template containing a member type, other, that produces allocators in the same family as X: if the name X is bound to SomeAllocator<value_type>, then rebind<U>::other is the same type as SomeAllocator<U>. The resulting type SameAllocator<U>::other shall meet the requirements of the Allocator concept. The concept map for Allocator<LegacyAllocator> maps rebind_type<U> to rebind<U>::other.

Rename rebind to rebind_type

Modify paragraph 4 to rename rebind:

```
typename generic_pointer;  
typename const_generic_pointer;
```

A type that can store value of a pointer (const_pointer) from any allocator in the same family (see member template rebind_type in 20.7.2.1) as X and which will produce the same value when explicitly converted back to that pointer type. For any two allocators X, and Y of the same family, the implementation of a library facility using Allocator<X> and Allocator<Y>, is permitted to add additional requirements, SameType<Allocator<X>::generic_pointer, Allocator<Y>::generic_pointer> and SameType<Allocator<X>::const_generic_pointer, Allocator<Y>::const_generic_pointer> [Example:

```
template<ObjectType T, Allocator Alloc = allocator<T> >  
requires Destructible<T> &&  
SameType<Alloc::generic_pointer,  
        Alloc::Rrebind_type<list_node<T>>::generic_pointer> &&  
SameType<Alloc::const_generic_pointer,  
        Alloc::Rrebind_type<list_node<T>>::const_generic_pointer>  
class list;
```

And the same thing for paragraph 7:

```
template<ObjectType T> class rebind_type;
```

Class Template: The associated template rebind_type is a template that produces allocators in the same family as X: if the name X is bound to SomeAllocator<value_type>, then rebind_type<U> is the same type as SomeAllocator<U>. The resulting type SameAllocator<U> shall meet the requirements of the Allocator concept. The default value for rebind_type is a template R for which R<U> is X::template rebind<U>::other.

Note that the last use of rebind is deliberate and should not be changed to rebind_type.

And also section 14.9.2.2 [concept.map.assoc], paragraph 3:

```
concept Allocator<typename Alloc> {  
    template<class T> class rebind;  
}
```

```

template<typename T>
class my_allocator {
    template<typename U> class rebind type;
};

template<typename T>
concept_map Allocator<my_allocator<T>> {
    template<class U>
        using rebind type = my_allocator<T>::rebind type;
}

```

Replace `construct_element` with `construct`

In section 20.7 [memory], in the synopsis for <memory>, remove `construct_element`:

```

//20.7.10, construct_element
template<Allocator Alloc, class T, class... Args>
requires AllocatableElement<Alloc, T, Args&&...>
void construct_element(Alloc& alloc, T& r, Args&&... args);

```

In section 20.7.3 [allocator.element.concepts] paragraph 8, modify the definition of the `AllocatableElement` concept and eliminate the related concept map:

```

auto concept AllocatableElement<class Alloc, classObjectType T, class...
Args>
{
    requires Allocator<Alloc>;
    requires FreeStoreAllocatable<T>;
    void Alloc::construct_element(Alloc& a, T* t, Args&&... args);

    void X::destroy(value type* p) {
        addressof(*p)->~value type();
    }
}

```

Remove section 20.7.10 entirely:

```

20.7.10 construct_element[construct_element]
template<Allocator Alloc, class T, class... Args>
requires AllocatableElement<Alloc, T, Args&&...>
void construct_element(Alloc& a, T& r, Args&&... args);

[ Note: The appropriate overload of the construct_element function is called from within containers to
construct elements during insertion operations and to move elements during reallocation operations. It
automates the process of determining whether the scoped allocator model is in use and transmitting the inner
allocator for scoped allocators. —end note—]

Effects: AllocatableElement<Alloc, T, Args&&...>::construct_element(a, addressof(r),
forward<Args>(args)...)


```

Modify Section 23.1.1 [container.requirements.general], paragraph 3 as follows:

For the components defined in this clause that declare an allocator type, objects stored in these components shall be constructed using ~~construct_element (20.7.10)~~ the construct member function, and destroyed using the destroy member function of the container's allocator (20.7.2.2) unless otherwise specified. (construct and destroy are specified in the AllocatableElement concept (20.7.3)). ~~A container may directly call constructors and destructors for its stored objects, without calling the construct_element or destroy functions, if the allocator models the MinimalAllocator concept.~~ *[Note: If the component is instantiated with a scoped allocator of type A (i.e., ~~an allocator that meets the requirements of the ScopedAllocator concept~~ one of scoped_allocator_adaptor or scoped_allocator_adaptor2), then construct_element may pass an inner allocator argument to T's constructor. — end note]*

Rename construct_element to construct in section 23.2.7 [vector.bool], paragraph 2:

Unless described below, all operations have the same requirements and semantics as the primary vector template, except that operations dealing with the bool value type map to bit values in the container storage and AllocatableElement::construct_element (23.1) (20.7.3) is not used to construct these values.

Allocator Propagation

At the end of section 20.7.2.2 [allocator.concepts], add descriptions of the propagation functions:

```
X select_on_container_copy_construction(const X& x);
```

Returns: x for allocators that should propagate from the existing container to the new container on copy-construction, X() otherwise.

Default: returns x

Remarks: Used to select the allocator for a new container during copy-construction. The choice as to whether the allocator should propagate from the existing container to the new one varies by allocator type. See 23.1.1 [container.requirements.general].

[Note: in situations where the copy constructor for a container is elided, this function is not called. The behavior in these cases is as if select_on_container_copy_construction returned x — end note]

```
X select_on_container_move_construction(X&& x)
```

Returns: move(x) for allocators that should propagate from the existing container to the new container on move-construction, X() otherwise.

Default behavior: returns move(x)

Remarks: Used to select the allocator for a new container during move-construction. The choice as to whether the allocator should propagate from the existing container to the new one varies by allocator type. See 23.1.1 [container.requirements.general].

[*Note:* in situations where the move constructor for a container is elided, this function is not called. The behavior in these cases is as if `select_on_container_move_construction` returned `x` – *end note*]

```
void do_on_container_copy_assignment(X& to, const X& from);
```

Effects: assign `to = from` for allocators that should propagate from the right-hand container to the left-hand container on container copy-assignment; otherwise no effect.

Default behavior: does nothing

Remarks: The choice as to whether the allocator should propagate from the right-hand container to the left-hand container during assignment one varies by allocator type. See 23.1.1 [container.requirements.general].

```
void do_on_container_move_assignment(X& to, X&& from) { }
```

Effects: assign `to = move(from)` for allocators that should propagate from the right-hand container to the left-hand container on container move-assignment; otherwise no effect.

Default behavior: does nothing

Remarks: The choice as to whether the allocator should propagate from the right-hand container to the left-hand container during assignment one varies by allocator type. See 23.1.1 [container.requirements.general].

```
void do_on_container_swap(X& a, X& b);
```

Effects: assign `swap(to, from)` for allocators that should propagate on container swap; otherwise no effect.

Default behavior: does nothing

Remarks: The choice as to whether the allocator should be swapped when containers are swapped varies by allocator type. See 23.1.1 [container.requirements.general].

In section 20.7 [memory], remove mention of the `allocator_propagate` traits from the `<memory>` synopsis:

```
// 20.7.4, allocation propagation traits
template <class Alloc> struct allocator_propagate_never;
template <class Alloc> struct allocator_propagate_on_copy_construction;
template <class Alloc> struct allocator_propagate_on_move_assignment;
template <class Alloc> struct allocator_propagate_on_copy_assignment;
template <class Alloc> struct allocator_propagation_map;
```

and

```
template <class Allocator OuterA, class Allocator InnerA>
struct allocator_propagate_never<scoped_allocator_adaptor<OuterA, InnerA>
>
: true_type { };
```

Remove section 20.7.4 [allocator.propagation] in its entirety:

20.7.4 Allocator Propagation Traits [allocator.propagation]

```
template <class Alloc> struct allocator_propagate_never  
—: false_type{ };  
  
etc...
```

Remove section 20.7.5 [allocator.propagation.map] in its entirety:

20.7.5 Allocator propagation map [allocator.propagation.map]

```
template <class Alloc> struct allocator_propagation_map {  
—static Alloc select_for_copy_construction(const Alloc&);  
—static void move_assign(Alloc& to, Alloc&& from);  
—static void copy_assign(Alloc& to, Alloc& from);  
—static void swap(Alloc& a, Alloc& b);  
}  
  
etc...
```

Modify the end of Section 23.1.1 [container.requirements.general], paragraph 4:

Notes: the algorithms `swap()`, `equal()` and `lexicographical_compare()` are defined in Clause 25. Those entries marked “(Note A)” should have constant complexity. Those entries marked “(Note B)” have constant complexity unless ~~`allocator_propagate_never<X::allocator_type>::value is true`~~ `Allocator<allocator_type>::select for move construction` returns an allocator different from `rv.get_allocator()`, in which case they have linear complexity.

Modify Section 23.1.1 [container.requirements.general], paragraph 8:

Unless otherwise specified, all containers defined in this clause obtain memory using an allocator (See 20.7.2). Copy and move constructors for all these container types ~~defined in this Clause~~ obtain an allocator by calling ~~`allocator_propagation_map::select_for_copy_construction()`~~ `Allocator<allocator_type>::select on container copy construction` or `Allocator<allocator_type>::select on container move construction` on their respective first parameters. All other constructors for these container types take an `Allocator` argument (20.1.2), an allocator whose value type is the same as the container’s value type. A copy of this argument is used for any memory allocation performed, by these constructors and by all member functions, during the lifetime of each container object or until the allocator is replaced. The allocator may be replaced only via assignment or `swap()`. Allocator replacement is performed by calling ~~`allocator_propagation_map<allocator_type>::move_assign()`, `allocator_propagation_map<allocator_type>::copy_assign()`, or `allocator_propagation_map<allocator_type>::swap()`~~ `Allocator<allocator_type>::do on container copy assignment`, `Allocator<allocator_type>::do on container move assignment`, or `Allocator<allocator_type>::do on container swap` within the implementation of the corresponding container operation. Calling the preceding `Allocator` functions may or may not modify the allocator, depending on the implementation of those functions for the specific allocator type. In all container types defined in this Clause, the member `get_allocator()` returns a copy of the allocator object used to construct the container, or most recently used to replace the allocator.

Remove `is_scoped_allocator` trait

In section 20.7 [memory], remove `is_scoped_allocator` from the <memory> synopsis:

~~// 20.7.3, allocator-related traits~~

~~template <class Alloc> struct is_scoped_allocator;~~

and

~~template <class Allocator OuterA, class Allocator InnerA>
struct is_scoped_allocator<scoped_allocator_adaptor<OuterA, InnerA>>
: true_type { };~~

Remove paragraphs 3 and 4 from section 20.7.3 [allocator.element.concepts]:

~~template <class Alloc> struct is_scoped_allocator : false_type { };~~

~~[Note: If a specialization `is_scoped_allocator<Alloc>` is derived from `true_type`, it indicates that `Alloc` is a scoped allocator. A scoped allocator specifies the memory resource to be used by a container (as all allocators do) and also specifies an inner allocator resource to be used by every element of the container. — end note]~~

~~Requires: If a specialization `is_scoped_allocator<Alloc>` is derived from `true_type`, `Alloc` shall have a nested type `inner_allocator_type` and a member function `inner_allocator()` which is callable with no arguments and which returns an object of a type that is convertible to `inner_allocator_type`.~~

Remove the last sentence of paragraph 2 from Section 23.1.4 [associative.reqmnts]:

Each associative container is parameterized on `Key` and an ordering relation `Compare` that induces a strict weak ordering (25.3) on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary type `T` with the `Key`. The object of type `Compare` is called the comparison object of a container. This comparison object may be a pointer to function or an object of a type with an appropriate function call operator. ~~If the `Compare` type uses an allocator, then it conforms to the same rules as a container item; the container will construct the comparison object with the allocator appropriate to the allocator-related traits of the `Compare` type and whether `is_scoped_allocator` is true for the container's allocator type.~~

Remove the last sentence of paragraph 3 in Section 23.1.5 [unord.req]:

Each unordered associative container is parameterized by `Key`, by a function object `Hash` that acts as a hash function for values of type `Key`, and by a binary predicate `Pred` that induces an equivalence relation on values of type `Key`. Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary mapped type `T` with the `Key`. ~~If the `Hash` and/or the `Pred` type use an allocator, then they conform to the same rules as container items; the container will construct the `Hash` and `Pred` objects with the allocator appropriate to the allocator-related traits of the `Hash` and `Pred` types and whether `is_scoped_allocator` is true for the container's allocator type.~~

Modified `scoped_allocator_adaptor`

Modify the introduction of section 20.7.7 [allocator.adaptor] as follows:

20.7.7 Scoped Allocator Adaptor [allocator.adaptor]

The `scoped_allocator_adaptor` and `scoped_allocator_adaptor2` class templates ~~is an~~ are allocator templates that specifies the memory resource (the *outer allocator*) to be used by a container (as any other allocator does) and also specifies an *inner allocator* resource to be used by every element in the container. ~~This adaptor is instantiated with outer and inner allocator types. If The `scoped_allocator_adaptor` template is instantiated with only one allocator type (i.e., the second type is void), the same allocator type is used for both the outer and inner allocator types and the same allocator instance is used for both the outer and inner allocator instances. The `scoped_allocator_adaptor2` template is instantiated with two allocator types, one for the inner allocator and one for the outer allocator. The interface is specialized for the single-allocator case such that it takes only one allocator instance argument in the constructor, versus two allocators for the general case. Otherwise, the interface to the specialized and general cases are the same. A `scoped_allocator_adaptor` that is instantiated with two identical parameters is different than an adaptor instantiated with only one parameter: the former may be constructed with different instances of outer and inner allocators whereas the second may be constructed only with one allocator instance. [Note: the `scoped_allocator_adaptor` and `scoped_allocator_adaptor2` are is derived from the outer allocator type, so it they can be substituted for the outer allocator type in most expressions. – end note]. To minimize the chance that an allocator will be constructed in an inappropriate scope, these adaptors are not propagated on copy and move construction unless instantiated with allocators that are not default-constructible.~~

In section 20.7.7 [allocator.adaptor], completely replace the declarations `scoped allocators` with the following:

```
// Base class for exposition only
template<Allocator Alloc>
class scoped_allocator_adaptor_base : public Alloc
{
public:
    typedef Alloc outer_allocator_type;

    typedef Allocator<Alloc>::size_type      size_type;
    typedef Allocator<Alloc>::difference_type difference_type;
    typedef Allocator<Alloc>::pointer        pointer;
    typedef Allocator<Alloc>::const_pointer  const_pointer;
    typedef Allocator<Alloc>::generic_pointer generic_pointer;
    typedef Allocator<Alloc>::const_generic_pointer
        const_generic_pointer;
    typedef Allocator<Alloc>::reference      reference;
    typedef Allocator<Alloc>::const_reference const_reference;
    typedef Allocator<Alloc>::value_type    value_type;

    pointer      address(reference x)      const;
    const_pointer address(const_reference x) const;

    pointer allocate(size_type n);
    pointer allocate(size_type n, const_generic_pointer u);
    void deallocate(pointer p, size_type n);
    size_type max_size() const;

    void destroy(value_type* p);
};
```

```

template<Allocator Alloc>
class scoped_allocator_adaptor
    : public scoped_allocator_adaptor_base<Alloc>
{
    typedef Alloc outer_allocator_type;
    typedef Alloc inner_allocator_type;

    requires DefaultConstructible<Alloc> scoped_allocator_adaptor();
    scoped_allocator_adaptor(scoped_allocator_adaptor&&);
    scoped_allocator_adaptor(const scoped_allocator_adaptor&);
    scoped_allocator_adaptor(Alloc&& outerAlloc);
    scoped_allocator_adaptor(const Alloc& outerAlloc);

    template <Allocator Alloc2>
        requires Convertible<Alloc2&&, Alloc>
        scoped_allocator_adaptor(scoped_allocator_adaptor<Alloc2>&&);
    template <Allocator Alloc2>
        requires Convertible<const Alloc2&, Alloc>
        scoped_allocator_adaptor(const scoped_allocator_adaptor<Alloc2>&);

    template <class T, class... Args>
        requires ConstructibleWithAllocator<T, inner_allocator_type, Args&&...>
        void construct(T* p, Args&&... args);

    //stop recursion
    template <class T, Allocator Alloc2, class... Args>
        requires ConstructibleWithAllocator<T, Alloc2, Args&&...>
        void construct(T* p, allocator_arg_t,
            const Alloc2&, Args&&... args);

    const Alloc& outer_allocator() const;
    const Alloc& inner_allocator() const;
};

template<Allocator OuterA, Allocator InnerA>
class scoped_allocator_adaptor2
    : public scoped_allocator_adaptor_base<OuterA>
{
public:
    typedef OuterA outer_allocator_type;
    typedef InnerA inner_allocator_type;

    requires DefaultConstructible<OuterA> && DefaultConstructible<InnerA>
        scoped_allocator_adaptor2();
    scoped_allocator_adaptor2(scoped_allocator_adaptor2&& other);
    scoped_allocator_adaptor2(const scoped_allocator_adaptor2& other);
    scoped_allocator_adaptor2(OuterA&& outerAlloc,
        InnerA&& innerAlloc);
    scoped_allocator_adaptor2(const OuterA& outerAlloc,
        const InnerA& innerAlloc);

```

```

template <Allocator OuterA2, Allocator InnerA2>
    requires Convertible<OuterA2&&, OuterA>
        && Convertible<InnerA2&&, InnerA>
    scoped_allocator_adaptor2(
        scoped_allocator_adaptor2<OuterA2, InnerA2>&&);
template <Allocator OuterA2, Allocator InnerA2>
    requires Convertible<const OuterA2&, OuterA>
        && Convertible<const InnerA2&, InnerA>
    scoped_allocator_adaptor2(
        const scoped_allocator_adaptor2<OuterA2, InnerA2>&);

template <class T, class... Args>
    requires ConstructibleWithAllocator<T, inner_allocator_type, Args&&...>
    void construct(T* p, Args&&... args);

// Recursion stop
template <class T, Allocator Alloc2, class... Args>
    requires ConstructibleWithAllocator<T, Alloc2, Args&&...>
    void construct(T* p, allocator_arg_t,
        const Alloc2&, Args&&... args);

const OuterA& outer_allocator() const;
const InnerA& inner_allocator() const;

private:
    inner_allocator_type inner_alloc; // for exposition only
};

template <Allocator Alloc1, Allocator Alloc2>
    bool operator==(const scoped_allocator_adaptor<Alloc1>& a,
        const scoped_allocator_adaptor<Alloc1>& b);
template <Allocator Alloc1, Allocator Alloc2>
    bool operator!=(const scoped_allocator_adaptor<Alloc1>& a,
        const scoped_allocator_adaptor<Alloc1>& b);

template <Allocator OuterA1, Allocator InnerA1,
    Allocator OuterA2, Allocator InnerA2>
    bool operator==(const scoped_allocator_adaptor2<OuterA1, InnerA1>& a,
        const scoped_allocator_adaptor2<OuterA1, InnerA1>& b);
template <Allocator OuterA1, Allocator InnerA1,
    Allocator OuterA2, Allocator InnerA2>
    bool operator!=(const scoped_allocator_adaptor2<OuterA1, InnerA1>& a,
        const scoped_allocator_adaptor2<OuterA1, InnerA1>& b);

template <Allocator A>
    concept_map Allocator<scoped_allocator_adaptor<A> > {

        typedef scoped_allocator_adaptor<A> X; // for exposition only

        typedef X::value_type          value_type;
        typedef X::pointer              pointer;

```

```

typedef X::const_pointer      const_pointer;
typedef X::generic_pointer    generic_pointer;
typedef X::const_generic_pointer const_generic_pointer;
typedef X::difference_type    difference_type;
typedef X::size_type          size_type;
typedef X::reference           reference;
typedef X::const_reference     const_reference;

template <class T> using rebind_type =
    scoped_allocator_adaptor<Allocator<A>::rebind_type<T> >;

requires DefaultConstructible<X>
    X select_on_container_copy_construction(const X&) { return X(); }
requires ! DefaultConstructible<X>
    X select_on_container_copy_construction(const X& x) { return x; }
requires DefaultConstructible<X>
    X select_on_container_move_construction(X&&) { return X(); }
requires ! DefaultConstructible<X>
    X&& select_on_container_move_construction(X&& x) { return move(x); }
}

template <Allocator A1, Allocator A2>
concept_map Allocator<scoped_allocator_adaptor2<A1, A2> > {
    typedef scoped_allocator_adaptor2<A1, A2> X; // for exposition only

    typedef X::value_type      value_type;
    typedef X::pointer          pointer;
    typedef X::const_pointer    const_pointer;
    typedef X::generic_pointer  generic_pointer;
    typedef X::const_generic_pointer const_generic_pointer;
    typedef X::difference_type  difference_type;
    typedef X::size_type        size_type;
    typedef X::reference         reference;
    typedef X::const_reference   const_reference;

    template <class T> using rebind_type =
        scoped_allocator_adaptor2<Allocator<A1>::rebind_type<T>, A2>;

    requires DefaultConstructible<X>
        X select_on_container_copy_construction(const X&) { return X(); }
    requires ! DefaultConstructible<X>
        X select_on_container_copy_construction(const X& x) { return x; }
    requires DefaultConstructible<X>
        X select_on_container_move_construction(X&&) { return X(); }
    requires ! DefaultConstructible<X>
        X&& select_on_container_move_construction(X&& x) { return move(x); }
}

```

Replace the function descriptions in sections 20.7.7.1 [allocator.adaptor.cntr], 20.7.7.2 [allocator.adaptor.members], and 20.7.7.3 [allocator.adaptor.globals] with the following:

20.7.7.1 `scoped_allocator_adaptor_base` members [`allocator.adaptor.base`]

```
pointer      address(reference x)      const;
const_pointer address(const_reference x) const;

returns: this->outer_allocator_type::address(x);

pointer allocate(size_type n);

returns: this->outer_allocator_type::allocate(n);

template <typename _HintP>
pointer allocate(size_type n, _HintP u);

returns: this->outer_allocator_type::allocate(n, u);

void deallocate(pointer p, size_type n);

effects: this->outer_allocator_type::deallocate(p, n);

size_type max_size() const;

returns: this->outer_allocator_type::max_size();

void destroy(pointer p);

effects: this->outer_allocator_type::destroy(p);
```

20.7.7.2 `scoped_allocator_adaptor` constructors [`allocator.adaptor.cntr`]

```
requires DefaultConstructible<Alloc> scoped_allocator_adaptor();
```

effects: Default-initializes the `Alloc` sub-object.

```
scoped_allocator_adaptor(scoped_allocator_adaptor&& other);
scoped_allocator_adaptor(const scoped_allocator_adaptor& other);
```

effects: initializes the `Alloc` sub-object from `other.outer_allocator()`.

```
scoped_allocator_adaptor(OuterA&& outerAlloc);
scoped_allocator_adaptor(const OuterA& outerAlloc);
```

effects: initializes the `Alloc` sub-object from `outerAlloc`.

```
template <Allocator Alloc2>
requires Convertible<Alloc2&&, Alloc>
scoped_allocator_adaptor(scoped_allocator_adaptor<Alloc2>&& x);
template <Allocator Alloc2>
requires Convertible<const Alloc2&, Alloc>
scoped_allocator_adaptor(const scoped_allocator_adaptor<Alloc2>& x);

effects: initializes the Alloc sub-object from x.outer_allocator().
```


20.7.7.3 `scoped_allocator_adaptor2` constructors [allocator.adaptor2.cntr]

```
requires DefaultConstructible<OuterA> && DefaultConstructible<InnerA>
scoped_allocator_adaptor2();
```

effects: Default-initializes the `OuterA` sub-object and `inner_alloc` member.

```
scoped_allocator_adaptor2(scoped_allocator_adaptor2&& other);
scoped_allocator_adaptor2(const scoped_allocator_adaptor2& other);
```

effects: initializes the `OuterA` sub-object from `other.outer_allocator()` and the `inner_alloc` member from `other.inner_allocator()`.

```
scoped_allocator_adaptor2(OuterA&& outerAlloc,
                          InnerA&& innerAlloc);
scoped_allocator_adaptor2(const OuterA& outerAlloc,
                          const InnerA& innerAlloc);
```

effects: initializes the `OuterA` sub-object from `outerAlloc` and the `inner_alloc` member from `innerAlloc`.

```
template <Allocator OuterA2, Allocator InnerA2>
requires Convertible<OuterA2&&, OuterA>
        && Convertible<InnerA2&&, InnerA>
scoped_allocator_adaptor2(
    scoped_allocator_adaptor2<OuterA2&, InnerA2>&& x);
template <Allocator OuterA2, Allocator InnerA2>
requires Convertible<const OuterA2&, OuterA>
        && Convertible<const InnerA2&, InnerA>
scoped_allocator_adaptor2(
    const scoped_allocator_adaptor2<OuterA2&, InnerA2>&x);
```

effects: initializes the `OuterA` sub-object from `x.outer_allocator()` and the `inner_alloc` member from `x.inner_allocator()`.

20.7.7.4 `scoped_allocator_adaptor` and `scoped_allocator_adaptor2` members [allocator.adaptor.members]

```
template <class T, class... Args>
requires ConstructibleWithAllocator<T, inner_allocator_type, Args&&...>
void construct(T* p, Args&&... args);
```

Effects: `outer_allocator().construct(p, allocator_arg_t, inner_allocator(), forward<Args>(args)...)`

```
template <class T, Allocator Alloc2, class... Args>
requires ConstructibleWithAllocator<T, Alloc2, Args&&...>
void construct(T* p, allocator_arg_t,
              const Alloc2& a2, Args&&... args);
```

Effects: `outer_allocator().construct(p, allocator_arg_t, a2, forward<Args>(args)...)`

[*Note:* this overloaded version of `construct` is to prevent recursion into ever-deeper inner allocators in the case where the outer allocator is itself a `scoped_allocator_adaptor`. – *end note*]

```
const outer_allocator_type& outer_allocator() const;
```

returns: the outer allocator used to construct this object (i.e., the Alloc sub-object).

```
const inner_allocator_type& inner_allocator() const;
```

returns: the inner allocator used to construct this object. For `scoped_allocator_adaptor`, returns the same reference as `outer_allocator()`. For `scoped_allocator_adaptor`, returns the `inner_alloc` member.

20.7.7.5 `scoped_allocator_adaptor` globals [allocator.adaptor.globals]

```
template <Allocator Alloc1, Allocator Alloc2>
bool operator==(const scoped_allocator_adaptor<Alloc1>& a,
                const scoped_allocator_adaptor<Alloc1>& b);
```

returns: `a.outer_allocator() == b.outer_allocator()`

```
template <Allocator Alloc1, Allocator Alloc2>
bool operator!=(const scoped_allocator_adaptor<Alloc1>& a,
                const scoped_allocator_adaptor<Alloc1>& b);
```

returns: `!(a == b)`.

```
template <Allocator OuterA1, Allocator InnerA1,
          Allocator OuterA2, Allocator InnerA2>
bool operator==(const scoped_allocator_adaptor2<OuterA1, InnerA1>& a,
                const scoped_allocator_adaptor2<OuterA1, InnerA1>& b);
```

returns: `a.outer_allocator() == b.outer_allocator() && a.inner_allocator() == b.inner_allocator()`.

```
template <Allocator OuterA1, Allocator InnerA1,
          Allocator OuterA2, Allocator InnerA2>
bool operator!=(const scoped_allocator_adaptor2<OuterA1, InnerA1>& a,
                const scoped_allocator_adaptor2<OuterA1, InnerA1>& b);
```

returns: `!(a == b)`.

References

All documents referenced here can be found at

<http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2008/>.

[N2768](#): Allocator Concepts, part 1 (revision 2)

[N2654](#): Allocator Concepts (Rev 1)

[N2554](#): The scoped allocator model (Rev 2)

[N2525](#): Allocator-specific move and swap

[N2621](#): Core Concepts for the C++0x Standard Library

[N2623](#): Concepts for the C++0x Standard Library: Containers