Doc No:
Date:
Author:

Defects and Proposed Resolutions for Allocator Concepts (Rev 2)

Contents

National Body Comments Addressed in this Paper

Summary

Changes from N2829 (Rev 1)

-
-
- AllocatableElement has_custom_construct
-

Changes from N2810

-
- pair
- vector string Vector basic_string
- AdvancedAllocator custom_construct Allocator
- rebind_type related_instance rebind Allocator

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).

Typographical and Editorial Errors

Description of Issue

Proposed Resolution

`Allocator` concept does not match all C++03 allocators

Description of Issue

```
Allocator          auto
                Allocator
    auto                    auto
                                    generic_pointer
construct                          construct
     pointer                T*
```

Proposed Resolution

```
auto              Allocator                          Allocator
                    auto

LegacyAllocator              concept_map   Allocator
```

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

Description of Issue

rebind Allocator rebind

Alloc::rebind<U>::other
Allocator<Alloc>::rebind<U> other
rebind template <class T>
rebind = MyAlloc<T>; template <class T> struct rebind {
typedef MyAlloc<T> other; };

Proposed Resolution

rebind Allocator
rebind_type retype related_type sibling sibling_allocator
rebind_type
you when you've used the wrong one
rebind_type
rebind

Construct Method is Limited to `value_type`

Description of Issue

construct Allocator
value_type
rebind_type
Node<T> alloc_ allocator_type
== Alloc<Node<T>> Node
T U
rebind_type

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

Alloc<T> construct

Proposed Resolution

construct emplace

```
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

construct_element

construct_element
construct

Proposed Resolution

construct_element construct_element
AllocatableElement c construct
Allocator AllocatableElement

construct
construct_element
construct destroy Allocator
AllocatableElement

construct

construct

Allocator Propagation Relies on Traits Instead of Concepts

Description of Issue

allocator_propagate_never
allocator_propagate_on_copy_construction
allocator_propagate_on_move_assignment
allocator_propagate_on_copy_assignment

allocator_propagation_map
Allocator

Proposed Resolution

Allocator

```
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) { }
```

Allocator

`is_scoped_allocator` Trait is not Used

Description of Issue

is_scoped_allocator

is_scoped_allocator –

Proposed Resolution

is_scoped_allocator construct
ConstructibleWithAllocator

construct AllocatableElement

`scoped_allocator_adaptor` has errors

Description of Issue

scoped_allocator_adaptor
Allocator

- void *unspecified allocator type*
- construct destroy pointer value_type*
  construct

- Allocator

Proposed Resolution

Two Types of `scoped_allocator_adaptor`s

Description of Issue

scoped_allocator_adaptor

scoped_allocator_adaptor

scoped_allocator_adaptor<A1, A1>
A1 scoped_allocator_adaptor<A1>
A1

Proposed Resolution

scoped_allocator_adaptor scoped_allocator_adaptor2

unspecified allocator type

Proposed Wording

Typographical and Editorial Corrections

<memory>

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

pointer::reference
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).

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

~~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~~

```
  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> >;~~
```

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

~~template <class R, Etc…~~

Revised Allocator Concept

auto Allocator

HasDereference rebind_type

```
// 20.7.2.2 Allocator concepts
auto concept Allocator<Ttypename AllocX> see below;
auto concept LegacyAllocator<typename X> see below
template <LegacyAllocator X> concept_map Allocator<X> see below
```

Allocator

```
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) { }
}
```

construct

```
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

Allocator

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;
}
```

```
  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> {}
```

```
      construct()
      _____________                construct
                                              variadic
```

value_type
construct CopyConstructible

construct construct

generic_pointer pointer void*
pointer
Allocator LegacyAllocator

Allocator<LegacyAllocator> Allocator

20.7.2.2.2 Allocator and LegacyAllocator members [allocator.concepts.members]

generic_pointer

```
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.

```
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 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`

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::~~R~~rebind_type<list_node<T>>::generic_pointer> &&
  SameType<Alloc::const_generic_pointer,
           Alloc::~~R~~rebind_type<list_node<T>>::const_generic_pointer>
    class list;
```

```
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`

```
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`

<memory> 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);
```

AllocatableElement

```
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();
    }

}
```

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)...)

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*]

`construct_element` `construct`

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

```
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].

allocator_propagate

```
<memory>
  // 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_propagate_map;


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

~~20.7.4 Allocator Propagation Traits [allocator.propagation]~~

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

~~etc…~~

~~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…*~~

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.

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*

`is_scoped_allocator` `<memory>`

~~*// 20.7.3, allocator-related traits*~~
~~`template <class Alloc> struct is_scoped_allocator;`~~

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

~~`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.~~

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.~~

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 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`

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, verses 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.

```
// 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);}
}
```

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